究極の8ビットCPUへの誘い

# はじめて読む 6809

星山浩樹 著　村瀬康治 監修

アスキー出版局

# はじめて読む
# 6809

星山　浩樹　著　　　村瀬　康治　監修

アスキー出版局

# 著者まえがき

BASICを卒業されたみなさんは〝マシン語〟という言葉にどこか神秘的なイメージを抱いていることと思います．確かに，すべてが数字で表されるマシン語は，冷淡で人間の思考と相容れない部分があります．しかし実際に触れてみれば，とてもホットで，私達を熱くする魅力を，そしてまた広大な宇宙をそこに発見することでしょう．

幸か不幸か最近のパーソナル・コンピュータは，電源を入れるとすぐにBASICが使えるようになっており，コンピュータのしくみやマシン語の存在を意識させません．これはある意味では正しい方向ではあるのですが，コンピュータの本質を理解する妨げになっていますし，少し高度なことをしようとするとすぐにBASICという壁に遮られてしまうのも事実です．そもそもコンピュータとは，

**マシン語のみを実行する**

ものであるため，BASICの厚いカーテンの外側からではその能力の100％を発揮させることは不可能です．BASICでは〝自由な処理ができない〟，〝処理速度がおそい〟など，いくつもの不満をあげることができますが，本書はこのような不満を持っているみなさんのための，マシン語およびコンピュータの，本当の意味での入門をお手伝いします．

本書は6809用のマシン語入門書ですが，そのタイトル『はじめて読む6809』からもわかるように，村瀬康治氏の執筆された『はじめて読むマシン語』の6809版であり，本書の内容は全般にわたり，原著『はじめて読むマシン語』に準拠しています．特に6章までのコンピュータの基本的な知識に関する解説では，6809の特徴的な部分を除き，原著と同じ内容構成になっています．また，7章以降の6809マシン語命令の解説においても，なるべく原著の〝わかりやすさ〟を反映するように努めました．しかし，筆者の力がおよばず至らぬ点があれば，今後読者の方々のご叱正をお願い申し上げます．

『はじめて読むマシン語』が，数多くの80系パーソナル・コンピュータ・ユーザーのマシン語の手引き書となったように，本書が6809パーソナル・コンピュータ・ユーザーにとっての手引き書となれば，筆者としてこれに勝る喜びはありません.

なお，本書は村瀬氏の監修によるものですが，本書の6809に関する解説およびプログラムについての責任は，すべて筆者の負うところです.

最後になりましたが，富士通株式会社，株式会社日立製作所には，機材の提供等で大変お世話になりました．ここに感謝の意を表します.

1984年11月　　星山浩樹

# 監修のことば

本書『はじめて読む6809』は，Z-80CPUを対象にした拙著『はじめて読むマシン語』を母体として，星山浩樹さんの素晴らしいリライトによって6809を対象としたマシン語の入門書として誕生しました．

「君はまだBASIC荒野をさまよっているのか」

これはその「はじめて読むマシン語」(1983年10月初版発行）に当初予定していた幻のサブタイトルです．結局これは，もう少しやさしい表現となり，「ほんとうのコンピュータと出逢うために」という副題で出版されましたが，どちらも，BASICを脱出しなければコンピュータの世界は開かれないことを言っています．

Z-80CPUを対象にした『はじめて読むマシン語』は，今までに何冊かのマシン語の入門書に挑戦したにもかかわらず挫折した人を含めて，「ほんとうのコンピュータ」を知ろうとしている多くの人たちへ，ソフトウェア自立への足掛かりとなる書を贈りたい，という気持ちから書いたもので，私としてもかなり綿密な構成をしたつもりです．幸いこの本は，アンケート葉書などからみてもかなり好評で，各方面で広く受け入れられており，発売以来すでに何万人もの読者が「BASICから脱出」する契機となった書であると自負しています．

ところが『はじめて読むマシン語』は，Z-80を対象にしているため，発売された当初から，富士通や日立などのパーソナル·コンピュータのユーザーを中心に，非常に多くの人たちから，6809用の『はじめて読むマシン語』を出してほしいという希望が寄せられていました．これは私の宿題であり，早くリライトしなければ，と思っていましたが，なかなか手を付ける余裕がないままでいたところ，星山浩樹さんの素晴らしいリライトによって，『はじめて読む6809』として実現することができました．

これは氏の6809に対する豊富な知識と，情熱と，独創性によるものであり，原書以上の出来栄えに，6809の一ファンとしても非常に喜んでいます．星山さんに深くお礼を申し上げます．どうもありがとうございました．

本書のタイトルは『はじめて読む6809』ですが，ただ6809のマシン語の入門書にとどまらず，広く「ほんとうのコンピュータ」との出逢いの書ともなるでしょう．6809パーソナル・コンピュータを前に，本書を手にした方は，またとない絶好の機会です．この機を逃さずに，BASICばかりでは知ることのできない，ほんとうのコンピュータの世界へ，勇気を出してぜひ踏み込んでください．

本書をしっかりと読み進めば，マシン語やCPUを中心にしたコンピュータの基本的な働きは，実に単純であることに気づくでしょう．

途中で挫折せず，本書を繰り返し最後まで読み進んでみてください．そのとき，あなたの前にコンピュータの世界が大きく開かれていることでしょう．

次の言葉は，『はじめて読むマシン語』の冒頭部に書いた，読者へのメッセージですが，この言葉を『はじめて読む6809』の読者へも，ぜひ贈りたいと思います．

> BASICの殻から一歩を踏み出し，マシン語 ──つまりはコンピュータの基礎を学ぼうとしている賢明な読者に，心から励ましの言葉を送ります．
> あなたと〝コンピュータ〟，そのほんとうの出逢いは本書から始まるのかも知れません．

1984年11月　　村瀬康治

# CONTENTS

イラストレーション：細田　雅亮

# 1

# コンピュータの内部にさわる

●まず最初に，マシン語を学習するための道具の使い方を説明します．

マシン語の世界を知るためにはいろいろな知識や概念を勉強しなくてはなりませんが，そのためには，マシン語ツールがどうしても必要なのです．

みなさんがコンピュータをさわり始めた頃のことを思い出してください．きっとBASICのプログラムの入力の仕方や，リストのとり方，プログラムの実行方法などを，見よう見まねで覚えたのだと思います．マシン語を始めるにしても，まずそれと同じことができなくてはなりません．

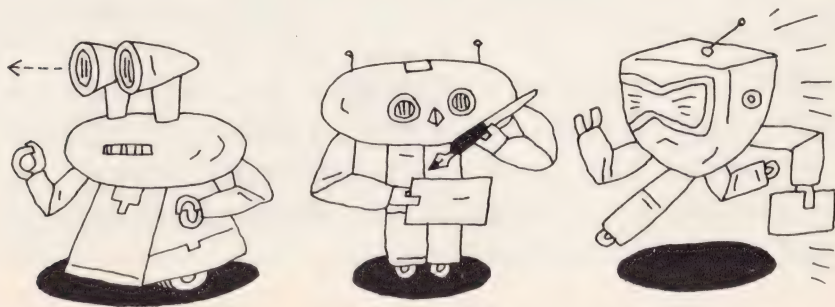
BASICでこういった操作を行う場合，LISTやRUNといった命令を使いますが，マシン語のレベルでは，モニタと呼ばれるプログラムが必要です．このプログラムは，メモリに書き込まれているデータを見たり，メモリに直接データを書き込むといったコンピュータの基本的な操作を行うものです．本書で扱うモニタは，みなさんのコンピュータに付いているものを利用しますので，本章の実習によってモニタの操作を十分マスターしてください．

# 1・1 モニタの機能

本書では，マシン語を理解するための道具として，パーソナル・コンピュータに付属しているモニタを利用しています．幸い6809を搭載したパーソナル・コンピュータでは，どの機種でも同じ機能のモニタを利用することができるので，次節の実習では，どの機種のユーザーでも例題どおりに実習することができます*1.

モニタには4つのコマンド(D, M, G, R)が付いていますが，本書で利用するコマンドは，次の3つです．

| | | |
|---|---|---|
| ① | Dコマンド | ……メモリの内容(メモリに記憶されているデータ)を見る |
| ② | Mコマンド | ……メモリに数値(データやプログラム)を書き込む |
| ③ | Gコマンド | ……プログラムを実行する |

このうち③の操作(Gコマンドの実習)では，各機種ごとに必要な準備がありますので，すでにモニタの操作を知っている人も③の実習だけは必ず読んでください．

*1 日立のS1のユーザーは，システムモード切換えスイッチによって，レベル3のモード(Bモード)で実習してください

# 1・2 モニタ操作の実習

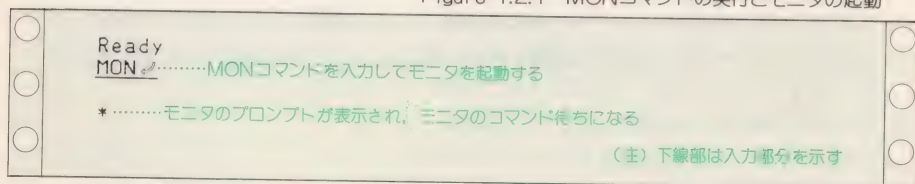
本節の目的は，あくまでもモニタの操作を覚えることですから，細かいことは気にする必要はありません．とにかく例題に従って操作してください．詳しい内容については，次章以降で解説します．

## “モニタの起動”

みなさんはBASICにMONというコマンドがあるのを知っていますか．このコマンドは本章で扱うモニタを起動するコマンドなのです．コンピュータの電源を入れてBASICが起動したら，Figure-1.2.1に示すようにMONコマンドを実行して，モニタモード(モニタが起動した状態)にはいってください．

Figure-1.2.1　MONコマンドの実行とモニタの起動

```
Ready
MON↵………MONコマンドを入力してモニタを起動する
*………モニタのプロンプトが表示され，モニタのコマンド待ちになる
```

（注）下線部は入力部分を示す

MONコマンドを実行するとモニタが起動され，プロンプト(＊)が表示されます．これは，モニタのコマンドを受け付ける準備ができていることを知らせる表示です．さあ，これでもうマシン語の世界にはいったのです．ここから先はBASICのコマンドは通用しません．この世界で通用するのは，これから学ぶモニタのコマンドだけなのです．

## メモリの内容を見る……Dコマンド

メモリの内容が見たいときは，Dコマンドを使います．Dに続けてアドレス*1を入力してリターンキーを押せば，そのアドレスから64番地分のメモリの内容が表示されます．

Dxxxx ⏎……xxxx番地から64番地分を表示する(下線部分を入力する)

Dの後ろのxxxxがアドレスですが，これは4桁の16進数*2で入力します．16進数については次章で詳しく説明しますので，〝8 E〞とか〝F F F E〞などが出てきても，「ああ，これは数字のことだな」と思っていてください．Figure-1.2.2を見ながら，とにかくDコマンドを実習してみましょう．実行した結果が，図のとおりになればOKです．

Figure-1.2.2 Dコマンドの実行例

```
*D0000
0000 02 86 78 00 00 00 00 00
0008 00 00 00 00 00 00 00 00
0010 00 22 22 00 16 00 00 04
0018 00 00 00 4F 00 00 05 78
0020 00 05 75 BA B8 EF B8 00
0028 00 00 00 01 00 0A 00 00
0030 00 70 96 0B F9 0E 5D 00
0038 00 00 00 0E 63 0E 63 70
*D
0040 A8 71 D4 71 D2 71 D4 FF
0048 FF 0C D3 00 00 00 00 03
0050 46 04 A6 0B F8 0E 5A 01
0058 05 46 0E 5F 79 04 00 00
0060 4D 00 00 00 00 00 00 00
0068 00 00 00 00 00 03 47 00
0070 0E 57 0C F1 00 00 00 00
0078 00 00 00 00 00 00 07 8D
*
```

アドレス表示部．それぞれの行の最初のメモリのアドレスが示される
$0000番地から64番地分のメモリの内容を表示する
$0000番地
$0006番地
1行には8バイトずつ表示される
$0007番地
$003F番地
アドレスを省略すると次の64番地分のメモリの内容を表示する
$0040番地
メモリの値は機種ごとに異なる
だいたいこのような表示をすればOK

*1 メモリに付けられた番地．詳しくは，2.1参照

*2 0～9，A～Fの数字を用いた記数法．16進数は10進数と区別するために数字の頭に〝$〞を付ける．詳しくは2.3参照

どうですか．期待どおりになったでしょうか？ ならなかった人はどこかやり方が違っているので，確認しながらもう一度試してみましょう．アドレスは4桁の数字(0～9，A～F)ですので，自分でいろいろなアドレスの内容をのぞいてみてください(どこをのぞいてもコンピュータは壊れません)．

## “メモリに数値(プログラムやデータ)を書く……Mコマンド”

メモリの内容を自由に見られるようになったら，こんどはメモリに新しい数値を書いてみましょう．メモリに数値を書き込むには，M コマンドを使います．このコマンドの入力方法も D コマンドと同様で，M に続けてアドレスを入力します．するとそのアドレスの内容が表示されて入力待ちになり，ここで新しい値を入力してやればそのアドレスのメモリは古い値が消えて，いま入力した新しい値になるのです．

| Mxxxx ⏎……xxxx 番地からメモリに数値を書き込む |
|---|

Figure-1.2.3 のとおりよく練習してください．

Figure-1.2.3　Mコマンドの実行例

```
*M6000⏎ ……$6000番地からメモリに書き込む
6000 00-4F⏎
6001 00-8E⏎
6002 00-60⏎
6003 00-20⏎
6004 00-A7⏎
6005 00-80⏎   もとのメモリのデータは，新しく入力したデータに変更される
6006 00-8B⏎
6007 00-11⏎
6008 00-24⏎
6009 00-FA⏎
600A 00-3F⏎
600B 00-⏎  ………値を書かずに⏎のみを入力するとそのアドレスの内容は変化しない
600C 00-.⏎ ………ピリオドを入力してMコマンドから抜ける
* ………モニタのコマンド待ちの状態に戻る
```

このようにMコマンドを使って，好きなアドレスに好きな値を書き込むことができるのです．この例以外にも自分で自由に試してください．ただし，書き換えてはならないアドレスもありますので，ここでは＄６０００番台(＄６０００～＄６ＦＦＦ)のアドレスだけで練習してください．

## “プログラムを実行する……Gコマンド”

マシン語のプログラムを実行するにはGコマンドを使います．このコマンドもDコマンドやMコマンドと同様，Gに続けてプログラムのアドレスを入力します．ただしこのアドレスは，プログラムのスタート・アドレス(プログラムの記憶されている最初のアドレス)を入力しなければなりません．

| Gxxxx ⏎……xxxx 番地からマシン語プログラムを実行する |
|---|

なお，このコマンドは非常に危険なコマンドなので，コマンドの実行には細心の注意が必要です．というのは，Gコマンドで誤ったマシン語のプログラムを実行すると，コンピュータはそのまま暴走して再びモニタに戻れなくなってしまうからです．BASICのプログラムを実行したときのように，親切なエラーメッセージは出力されません．

それでは，これからGコマンドを実行する準備にとりかかりましょう．まず，これまでに学んだ2つのコマンド(D, M)を使ってメモリの内容を書き換えます．この書き換えるメモリのアドレスは機種ごとに異なっているので，Figure-1.2.4の機種別アドレスを参照して，誤りのないように設定してください．Figure-1.2.5がメモリを書き換えている様子です．

| 機　　種 | 変更を行うアドレス | 書き込む値 |
|---|---|---|
| 日立 LEVEL-3(Mark II/V, S1のBモードを含む) | $0106～$0108 | $7E, $DC, $F0 |
| 富士通 FM-8 | $01D7～$01D9 | $7E, $AF, $A7 |
| 富士通 FM-7(FM-NEW7, FM-77を含む) | $01D7～$01D9 | $7E, $AB, $F9 |

Figure-1.2.4　Gコマンドのための初期設定(機種別アドレスと書き込む値)

Figure-1.2.5　機種別の初期設定の実行

Mコマンドでメモリの内容を変更した後は，必ずDコマンドでメモリの内容を確認してください．なお，今後この準備(プログラムを実行するための初期設定)は，コンピュータの電源を入れてモニタモードにはいったら毎回行う必要があります．操作の意味をここでは説明しませんので，しばらくは〝おまじない″だと思っていてください[*1]．

次に，実行するマシン語のプログラムを用意しなければなりません．実はさきほどMコマンドで＄６０００番地から入力した数値(16進数)の並びは，ある動作をする簡単なマシン語のプログラムです．そこでGコマンドの練習として，このプログラムを実行してみます．まずDコマンドで＄６０００番地以降のメモリの内容を表示して，Figure-1.2.3で入力したプログラムと同じかどうか確かめてください(Figure-1.2.6)．

＊1 機種別の初期設定についてはAPPENDIXを参照

Figure-1.2.6 Dコマンドでメモリに書き込んだプログラムを確認する

```
*D6000↵ ……Dコマンドで書き込んだメモリの内容を表示する

6000 4F 8E 60 20 A7 80 8B 11 ――Mコマンドでメモリに書き込んだプログラム
6008 24 FA 3F 00 00 00 00 00
6010 00 00 00 00 00 00 00 00
6018 00 00 00 00 00 00 00 00
6020 00 00 00 00 00 00 00 00
6028 00 00 00 00 00 00 00 00
6030 00 00 00 00 00 00 00 00
6038 00 00 00 00 00 00 00 00
*
```

プログラムの実行および結果の確認をしたのが Figure-1.2.7 です．図のように，コマンドを入力するとマシン語のプログラムの実行は一瞬で終わり，すぐにモニタのプロンプトが表示されます．

Figure-1.2.7 Gコマンドの実行と実行結果の確認

```
*G6000↵ ……Gコマンドでプログラムを実行する．プログラムは一瞬で終了する

*D6000↵ ……Dコマンドでプログラムの実行結果を確認する

6000 4F 8E 60 20 A7 80 8B 11 ――$6000～$600A番地はプログラム
6008 24 FA 3F 00 00 00 00 00
6010 00 00 00 00 00 00 00 00
6018 00 00 00 00 00 00 00 00
6020 00 11 22 33 44 55 66 77 ――$6020番地からのメモリがこのようになっ
6028 88 99 AA BB CC DD EE FF     ていればプログラムは正しく実行されている
6030 00 00 00 00 00 00 00 00
6038 00 00 00 00 00 00 00 00
*
```

図と同じ結果が得られましたか．＄６０２０番地からのメモリの内容が，

００，１１，２２，３３，……，ＤＤ，ＥＥ，ＦＦ

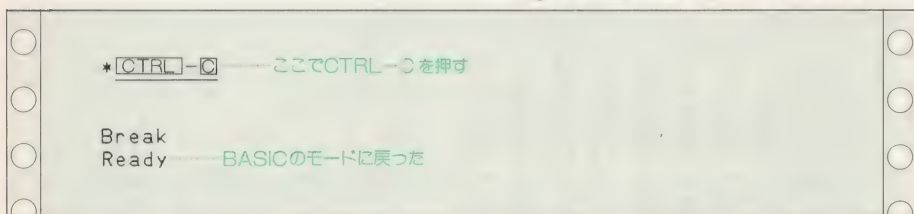
となっていればOKです．

もし，このようにならなかったら，Mコマンドで入力したプログラムをよく見直してください．また，いつまでたってもモニタのプロンプトが表示されなかったら，機種別の初期設定のアドレス(Figure-1.2.4)も確認する必要があります．

## “モニタの終了……CTRL-C”

CTRL-C……モニタを終了してBASICへ戻る

モニタを終了してBASICのモードに戻るには，CTRL-C( CTRL キーと C を同時に押す)を使います．リセットキーを押してもBASICのモードに戻ることはできますが，プログラムが暴走しない限りはCTRL-Cを使ってください．Figure-1.2.8を見るとわかるように“Ready”が表示され，確かにBASICのモードに戻っています．

Figure-1.2.8 モニタ・プログラムの終了

ここまでのことができるようになれば，あなたはもうマシン語を学ぶ道具を手に入れたことになります．3つのコマンドと初期設定を忘れずに，自信を持って読み進んでください．

# 2

## メモリの基礎知識

●前章では，マシン語を学ぶ道具であるモニタ操作の実習を行いましたが，モニタ・コマンドの実行によって，私たちは〝メモリの内容を見る〟，〝メモリの内容を書き換える〟，〝メモリに書き込んだプログラムを実行する〟ということを確認しました．このことから，すでにみなさんは〝メモリとはプログラムやデータを記憶するところ〟という概念を持っていると思います．そこで本章では，メモリに関するビット，バイト，アドレスといった概念をさらに具体的に解説していくことにします．これらの概念を理解するためには，２進数や16進数といった知識と深い関係がありますが，まず最初は，メモリの実態を把握し，その後にこれらの説明を行います．解説の順序が前後しているように思われるかもしれませんが，本章を通読した後で，再度本章の前半部分を読めば，さらに理解が深まると思います．

# 2-1 ビット，バイト，アドレス

メモリに関する概念としては，まず次の3つを理解しなければなりません．

| | |
|---|---|
| ビット……… | コンピュータが扱うデータの最小単位 |
| バイト……… | 8ビットを1バイトとする．1バイトのデータは，1つのメモリに記憶されるデータの単位 |
| アドレス…… | メモリを指定するための番地 |

これらの3つの概念は，マシン語レベルでコンピュータを操作するための最も基本的な概念です．本節では，これらを具体的に解説していきます．

## “ビット(Bit)”

コンピュータの内部で扱うデータは，すべて電圧が高いか，低いかの状態で表されています．電圧の高い状態は〝+5V″，低い状態は〝0V″になっているので，この2つの状態を電圧が〝ある″か〝ない″かで区別しているのです．そこで，〝ある″という状態を〝1″に，〝ない″という状態を〝0″に対応させて，1と0でデータを表現します．つまり，コンピュータの内部のデータは1と0の2つの値の組合せによって表され，メモリもCPUもすべて1と0で構成されるデータによって動作するのです．

この1と0はコンピュータで処理されるデータの最小単位であり，これ以上細かく分割することはできません．この単位を**ビット**と呼び，1ビットのデータは，1か0のいずれかの値をとります．つまり，1ビットは，1，0の2($2^1$)通りの状態を表すことができるのです．

# バイト(Byte)

ビットはコンピュータが扱うデータの最小単位ですが，CPU とメモリ間で行われるデータの処理は，8 ビットを基本単位として行われます．例えばメモリにプログラムやデータを書き込む場合，1 つのメモリには 8 ビットのデータが書き込まれますし，CPU がメモリからプログラムやデータを読み出して実行するときも，8 ビット単位で命令を取り込みます．そのため 8 ビットのデータは 1 つの単位として扱われ，それを**1 バイト**と呼びます．

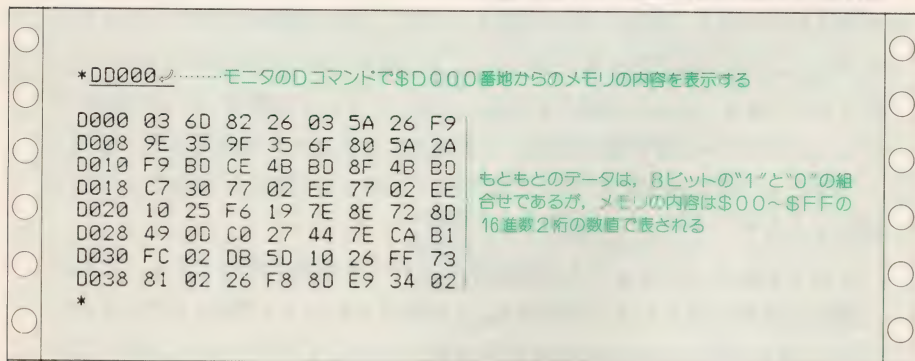
ビットが 8 つ集まることによって 1 バイトが構成されるのですから，1 バイトのビットパターンは次のように表されます．

00000000，00000001，00000010，00000011，00000100，00000101，
00000110，…………，11111100，11111101，11111110，11111111

1 ビットの場合に 2 通りの数が表せたように，8 ビットでは 256($2^8$)通りの数値を表すことができます．これは，16 進数で表すと＄００から＄ＦＦまでの 2 桁の数値となり，Figure-2.1.1 のモニタのＤコマンドで表示したメモリの内容と同じことが確認できます．どうして＄００から＄ＦＦまでとなるかについては，のちほど詳しく説明します．

Figure-2.1.1　16進数で表されたメモリの内容

```
*DD000↲……モニタのDコマンドで$D000番地からのメモリの内容を表示する

D000 03 6D 82 26 03 5A 26 F9
D008 9E 35 9F 35 6F 80 5A 2A
D010 F9 BD CE 4B BD 8F 4B BD
D018 C7 30 77 02 EE 77 02 EE
D020 10 25 F6 19 7E 8E 72 8D
D028 49 0D C0 27 44 7E CA B1
D030 FC 02 DB 5D 10 26 FF 73
D038 81 02 26 F8 8D E9 34 02
*
```

もともとのデータは，8ビットの“1”と“0”の組合せであるが，メモリの内容は＄００～＄ＦＦの16進数２桁の数値で表される

## “アドレス(Address)”

**アドレス**とは，何万個もあるメモリのなかから特定の1つのメモリを選択するために用いられる番地のことです．個々のメモリには1バイトのデータを記憶することができますが，その1バイトごとに1つのアドレスが割り当てられています．アドレスには16ビットのデータが用いられているので，ビットパターンでは，

0000000000000000～1111111111111111

のように表されます．16ビットのビットパターンでは，全部で65536($2^{16}$)通りの数値を表すことができるので，65536個のメモリを指定することができます．アドレスが16進数で表された場合には，＄0000番地から始まり(1番地からではない)，＄FFFF番地(＄FFFFは10進数で65535)までで表されます．このことを「6809のCPUのアドレス空間は＄0000番地から＄FFFF番地である」というような表現をします．

さきほどのFigure-2.1.1で，アドレスの表示部分が16進数の4桁で表されていることを確認してください(8ビットが1バイトであるからアドレスはちょうど2バイトで表すことができる)．

ビットとかバイトのほかに，よくキロビットとかキロバイトという表現が出てきます．これはそれぞれKビット，Kバイトのことで，私たちがキロメートルとかキログラムというのとよく似ています．しかし，マシン語の世界では，

**キロ(K)＝1024(＝$2^{10}$)**

の意味で，普通の〝キロ＝1000〟と少し違います．ですから64Kバイトといえば，64000バイトではなく〝64 * 1024＝65536〟バイトのことなのです．

それでは,いままで説明してきたビット,バイト,アドレスの関係を Figure-2.1.2 に示しましょう．もし 16 進数と 2 進数についての知識が必要であれば，2.3 節を随時参照してください．

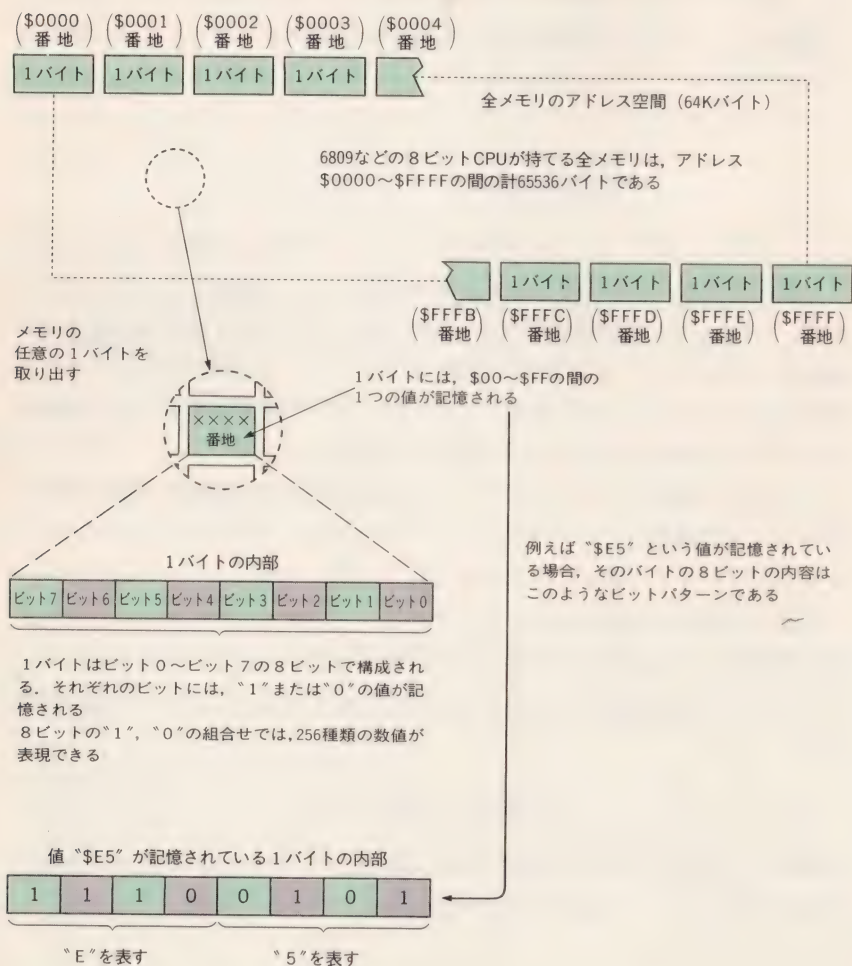



Figure-2.1.2 アドレス，バイト，ビット，「メモリ」の概念図

# 2.2 メモリの内容を見る

ビット，バイト，アドレスといった概念を明確にするために，メモリに関して具体的な解説を行っていきます．まず，Figure-2.2.1のプログラムを入力してください．

Figure-2.2.1　全アドレス空間表示プログラム

```
1000 DEF FNH$(X,N)=RIGHT$(STRING$(N-1,"0")+HEX$(X),N)+" "
1010 FOR I=0 TO 65536!/8-1
1020   PRINT FNH$(I*8,4);
1030   FOR J=0 TO 7
1040     PRINT FNH$(PEEK(I*8+J),2);
1050   NEXT
1060   PRINT
1070 NEXT
```

64Kバイトのアドレス空間をすべて表示するプログラム

このプログラムは，6809などの8ビットCPUを使ったコンピュータのすべてのアドレス空間(＄００００～＄ＦＦＦＦ)を連続的にスクリーンに表示するプログラムです．モニタのDコマンドでは一度に64バイトしか表示できませんが，このプログラムでFigure-2.1.2に示した64K(65536)バイトのアドレス空間を実際に確認してみることにしましょう．

Figure-2.2.2がこのプログラムの実行例です．＄００００番地から始まり＄ＦＦＦＦ番地まで表示が続けられますが，プログラムが終了するのにかなり時間がかかりますから途中でBREAKキーを押して，適当なところで中断してください．

Figure-2.2.2　全メモリアドレスのダンプリスト

Figure-2.2.2 のダンプリストは，FM-77 の例なので，＄０００0番地からはユーザーエリアの内容，＄８０００番地からは F-BASIC のインタープリタが表示されますが，ここでは，メモリ領域が何に使われているかは問題ではありません．左端のアドレス表示部やメモリの内容を表す 16 進数に注意してください．次々と表示されていくアドレスやデータを眺めているうちに，メモリの概念をある程度実感できるのではないでしょうか．Figure-2.2.3 は，ダンプリストの＄８０００番地以降を図示したものです．

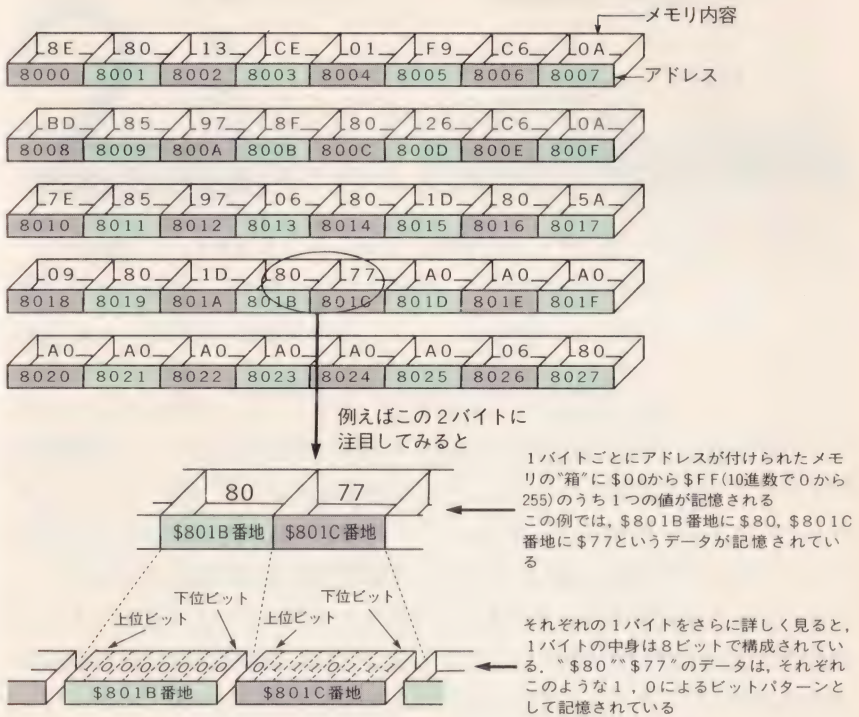



Figure-2.2.3 メモリの概念図

Figure-2.2.2のダンプリストやFigure-2.2.3を見ると，1つのメモリには1バイトのデータが記憶され，そのメモリを指し示すアドレスは2バイトの数値によって表されていることがわかりますが，この点についてもう少し詳しく説明しておきましょう．

例えば，Figure-2.2.2のダンプリストの＄８０３４番地のメモリの内容は＄ＣＥとなっていますが，これらの16進数をビットパターンで表してみましょう(Figure-2.2.4)．

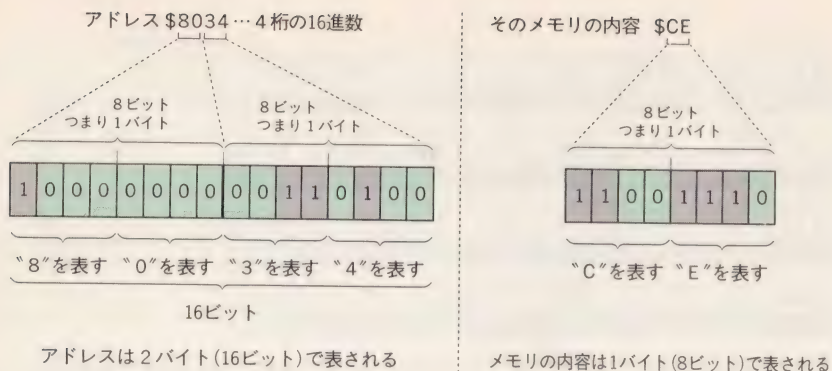



Figure-2.2.4 アドレスとメモリ内容の図解

この図から，アドレスは2バイト(16ビット)で表され，メモリの内容は1バイト(8ビット)で表されていることがわかると思います．CPUは，この16ビットのアドレス情報によって64Kバイトのアドレス空間のなかから該当する1つのメモリを捜し出し，そのメモリに記憶されている1バイトデータの読み出しまたは書き込みを行うことによって，プログラムを実行しているのです．これは6809や6800だけではなく，モトローラ系以外の6502，8080，8085，Z80などの8ビットCPUも，

**アドレス……16ビット**
**データ………8ビット**

で動作しています．

このように，8ビットのデータを1つの単位として処理するようなコンピュータを"8ビットのコンピュータ"といい，その2倍の16ビットのデータを1つの単位として処理できるコンピュータを"16ビットコンピュータ"といいます．これらの動作の仕組みについては，4章以降でさらに詳しく説明していきます．

# 2・3 16進数

ビット，バイト，アドレスの用語の説明では，すべて1，0で表された2進数でデータを表していましたが，単なる1，0のビットパターンを意味あるデータとしては理解しにくいものです．そのため，私たちがメモリにプログラムやデータを書き込んだり，メモリの内容を表示する場合には，いままでにも見てきたように0からFまでの数字を用いた**16進数**を使います．何も16進数を使わなくとも日頃使い慣れた10進数を使えばよいと思われるかもしれませんが，残念ながら，10進数はコンピュータの内部で処理するデータを表すには適していません．

16進数では，16種類の数字が使われます．0から9は10進数と同じものを用いますが，足りない6種類の数字には，A，B，C，D，E，Fのアルファベットを使います．16進数だからといってその数え方が特に難しいわけではなく，10進数との対応関係さえわかれば問題ありません．それでは，10進数と16進数の数の対応関係を見ながら8ビットのデータが表せるところまで順に数えていきましょう．カッコの中が10進数です．

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0<br>(0) | 1<br>(1) | 2<br>(2) | 3<br>(3) | 4<br>(4) | 5<br>(5) | 6<br>(6) | 7<br>(7) |
| 8<br>(8) | 9<br>(9) | A<br>(10) | B<br>(11) | C<br>(12) | D<br>(13) | E<br>(14) | F<br>(15) |
| 10<br>(16) | 11<br>(17) | 12<br>(18) | 13<br>(19) | 14<br>(20) | 15<br>(21) | 16<br>(22) | 17<br>(23) |
| 18<br>(24) | 19<br>(25) | 1A<br>(26) | 1B<br>(27) | 1C<br>(28) | 1D<br>(29) | 1E<br>(30) | 1F<br>(31) |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| F0<br>(240) | F1<br>(241) | F2<br>(242) | F3<br>(243) | F4<br>(244) | F5<br>(245) | F6<br>(246) | F7<br>(247) |
| F8<br>(248) | F9<br>(249) | FA<br>(250) | FB<br>(251) | FC<br>(252) | FD<br>(253) | FE<br>(254) | FF<br>(255) |

途中の数字は省略してありますが，モニタのDコマンドで表示したメモリの内容と同じ数え方です．

この例で示されているように，10進数の0から15(16進数では＄0から＄F)までの数が1桁で表され，同様に255(16進数で＄FF)までが2桁で表されていることに注意してください．8ビット(1バイト)のデータで表される数値の範囲は，16進数ではちょうど2桁で表せることになります．つまり，8ビットの2進数を表す表現法として，16進数はたいへん適した方法であることがわかります．

それでは次に，2進数で表されたデータと16進数の対応関係を見ていくことにしましょう．

## “2進数と16進数”

1と0で表されたビットパターンと16進数の間には一見何の関係も見出せないようですが，実は両者は密接な関係を持っているのです．まず，Figure-2.3.1の8ビットデータを例にして，16進数との対応関係を見ていくことにしましょう．

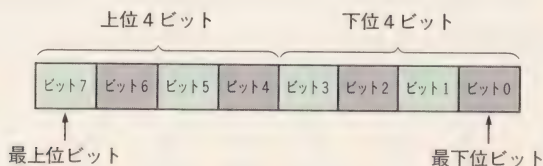



Figure-2.3.1　1バイトのビットパターン

この図で示したように，それぞれのビットはビット0，ビット1，…，ビット7というように呼ばれます．特に右端のビット0を最下位ビット，左端のビットを最上位ビットと呼びます．

2進数で表されたデータを16進数に変換するには，最下位ビットから数えて4ビットずつに分け，それぞれを16進数の1桁として扱います．この例で

は，8 ビットのデータを考えていますから，ちょうど真中から 4 ビットずつに分けられ，16 進数の 2 桁で表されるのです．ですから，2 進数の 4 ビットと 16 進数の 1 桁（＄0〜＄F）の対応関係さえわかってしまえば，16 進数⟷2 進数の変換は簡単に行うことができるのです．Figure-2.3.2 を見てください．この表には，2 進数，10 進数，16 進数の対応関係が示されています．

| 10進数 | 16進数 | 2進数の4ビット | | | |
|---|---|---|---|---|---|
| | | ビット 3 | ビット 2 | ビット 1 | ビット 0 |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 |
| 2 | 2 | 0 | 0 | 1 | 0 |
| 3 | 3 | 0 | 0 | 1 | 1 |
| 4 | 4 | 0 | 1 | 0 | 0 |
| 5 | 5 | 0 | 1 | 0 | 1 |
| 6 | 6 | 0 | 1 | 1 | 0 |
| 7 | 7 | 0 | 1 | 1 | 1 |
| 8 | 8 | 1 | 0 | 0 | 0 |
| 9 | 9 | 1 | 0 | 0 | 1 |
| 10 | A | 1 | 0 | 1 | 0 |
| 11 | B | 1 | 0 | 1 | 1 |
| 12 | C | 1 | 1 | 0 | 0 |
| 13 | D | 1 | 1 | 0 | 1 |
| 14 | E | 1 | 1 | 1 | 0 |
| 15 | F | 1 | 1 | 1 | 1 |

Figure-2.3.2 10進数，16進数，2進数の対応表

それでは, Figure-2.3.2 を見ながら, 適当な 16 進数を 2 進数に変換してみましょう.

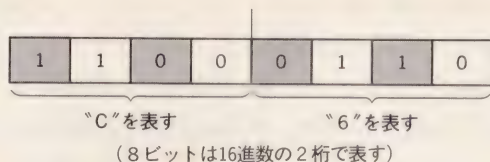



Figure-2.3.3 16進数の"$C6"のビットパターン

この図は, 16 進数の $ C 6 を 2 進数に直したものですが, 16 進数と 2 進数の関係は, Figure-2.3.4 のように考えるとすぐに理解できると思います.

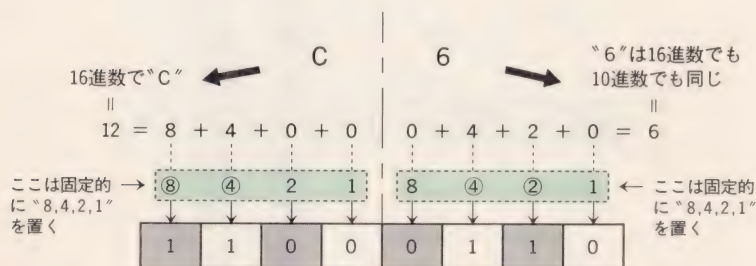
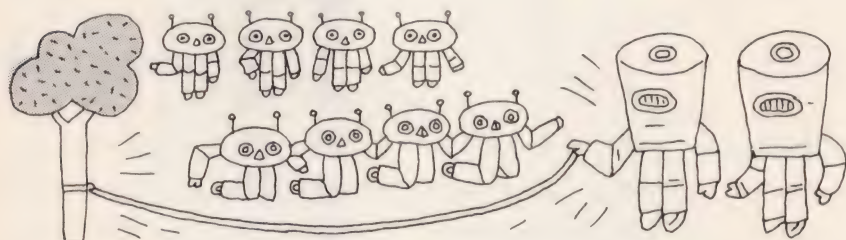



Figure-2.3.4 16進数と2進数の関係

この図の上位, 下位の 4 ビットに対して, それぞれ "8, 4, 2, 1" という数値が置かれていることに注意してください. 8 ビットのデータがあるとき, それを上位 4 ビット, 下位 4 ビットに分け, それぞれのビットに "8, 4, 2, 1" の数値を固定的に持たせます. そして, ビットパターンの "1" が書いてある

ビットの数値だけを4ビットごとに合計します．その合計された値が4ビットの2進数を10進数で表した値になりますから，Figure-2.3.2を見れば16進数を求めることができます．

具体例をいくつか見てみましょう．〝0101〟という4ビットを考えてみると，

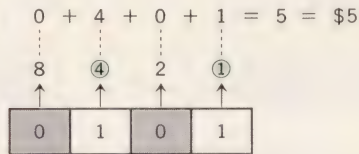



Figure-2.3.5　2進数から16進数へ ①

合計された10進数の値は5になり，これは16進数でも同じ$5になります．

もう1つ〝1010〟についても試してみましょう．

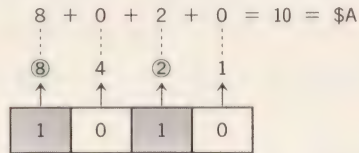
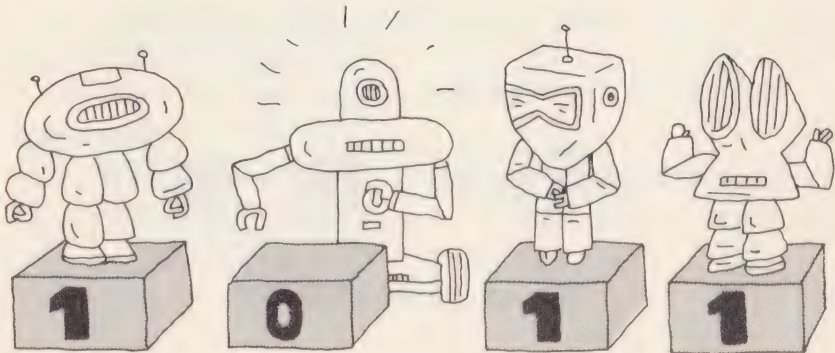



Figure-2.3.6　2進数から16進数へ ②

10進数の10は16進数では$Aになります．

いまの 2 つの例では, 2 進数から 16 進数への変換方法を見たわけですが, 逆の 16 進数から 2 進数への変換も原理的には同じように考えることができます. これもいくつか例をあげておきます.

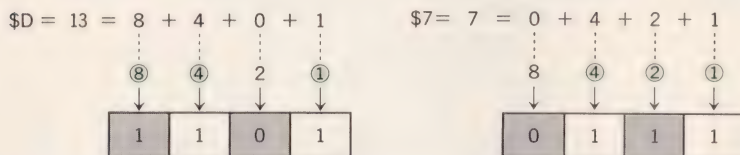



Figure-2.3.7 16進数から2進数へ

いままでの説明のキーポイントは, 各ビットの上に固定的に置かれた "8, 4, 2, 1" です. これさえ覚えていれば, 2 進数→ 16 進数の変換は, 1 になっているビットの上に置かれた数値を合計すれば 10 進数になり, さらに 16 進数にも変換できるのです. また, 16 進数→ 2 進数の変換も 16 進数をいったん 10 進数に直した後, "8, 4, 2, 1" のどれを取り上げて合計すれば 10 進数の合計と一致するかを考え, 取り上げたビットの位置に 1 を置き, 残りを 0 にします.

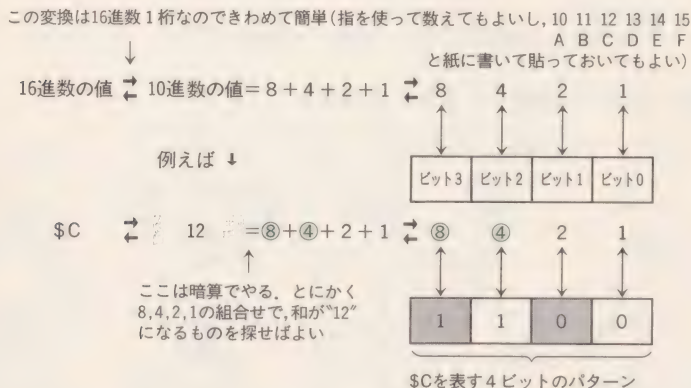



Figure-2.3.8 2進数, 16進数の相互変換

〝8，4，2，1〞という数字の種明かしをすると，これは〝$2^3$，$2^2$，$2^1$，$2^0$〞という意味です．一般に，nビットの2進数を10進数に変換する場合には，$a_i$をビットiとすると，

$$a_{n-1} \cdot 2^{n-1} + a_{n-2} \cdot 2^{n-2} + \cdots + a_0 \cdot 2^0$$

という式で計算するのですが，4ビットの2進数の変換にはこのような式を考える必要はないでしょう．〝8，4，2，1〞で十分です．

また，4ビット単位で考える2進数←→16進数の関係は，慣れてしまえばいちいちこのような変換作業は必要なくなります．4ビットのパターンを見た瞬間に〝1101〞は＄D，逆に＄Aなら〝1010〞というように，自然に対応させることができます．

どのような方法をとるにしろ，これらの16進数←→2進数の変換方法を理解していないとマシン語を扱うことはできません．

それでは，もう一度Figure-2.3.3へ戻って16進数，10進数，2進数の関係を再確認してみることにしましょう．次の図の意味はもう理解できると思います．

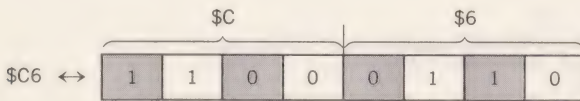



Figure-2.3.9　2進数，16進数の対応

これまでの説明から，2進数，16進数の数え方や8ビットのデータの表し方などが，ある程度明らかになったことと思います．なぜマシン語のレベルではデータが16進数で表されるのか，本章の最初に戻って読み返せば，さらに理解を深めることができるでしょう．

なお，参考までにBASICで行う10進数←→16進数の変換方法をFigure-2.3.10，2.3.11に紹介しておきましょう．この方法を用いれば，複数桁の10進数←→16進数の変換を簡単に行うことができます．

Figure-2.3.10　10進数→16進数の変換実行例

```
Ready
PRINT HEX$(15)↵
F
Ready
PRINT HEX$(16)↵
10
Ready
PRINT HEX$(100)↵
64
Ready
PRINT HEX$(1024)↵
400
Ready
PRINT HEX$(65535)↵
FFFF
Ready
```

10進数

16進数

プログラムを組む必要はなくダイレクトモードで実行する

Figure-2.3.11　16進数→10進数の変換プログラム実行例

16進数→10進数変換プログラム

```
10 INPUT A$
20 A=VAL("&H"+A$)
30 IF A<0 THEN A=A+2^16
40 PRINT A
50 GOTO 10
```

その実行例

```
Ready
RUN
? 0A↵
 10
? 0F↵
 15
? 10↵
 16
? FF↵
 255
? 3FF↵
 1023
? C000↵
 49152
? FFFF↵
 65535
?
```

16進数

10進数

# 3

# コンピュータはマシン語で動く

●最近のパーソナル・コンピュータは，電源を入れた時点でBASICが利用できるという，たいへん便利な環境を提供しています．しかし，コンピュータの本当の働きを理解しようとする場合，逆に本質的な意味をわかりにくくしているともいえるのです．

本章の目的は，コンピュータとマシン語の関係，つまりコンピュータはマシン語でなければ動かないという事実を説明することです．BASICのプログラムがどのように実行されているのか，また，CPUがマシン語のプログラムを実行した結果などを，例題を交えながら説明していきます．しかし，この時点では完全に理解することは難しいかもしれません．できれば本書を通読した後に，再度本章を読み返してください．ここで述べたことがさらによくわかると思います．

# 3・1 BASICインタープリタはマシン語のプログラム

私たちが日頃ちょっとした計算にコンピュータを使うのであれば，

PRINT　1+2

で答えが得られます．これはBASICを使い慣れた私たちにはしごく当然のことのように思えますが，実はそうではありません．なぜなら，コンピュータに対して通用する言葉は，本来はマシン語だけだからです．BASICで書いたプログラムは，結果としてコンピュータを働かせますが，それはあくまで結果であり，BASICで書いたプログラム自身では決してコンピュータは，働きません．電源を入れた時点でBASICが使える環境に慣れているユーザーにとって，コンピュータが働く本質的な意味が理解しにくくなっているのは仕方のないことかもしれません．しかし，「コンピュータはマシン語でしか働かない」という事実は，しっかり認識しておかなければなりません．

そうはいっても，BASICのプログラムで望みの処理を行わせることができるのも事実ですから，まずその理由を説明しておきましょう．

この一見矛盾に思えることを理解するためには，**BASICインタープリタ**の存在を知らなければなりません．BASICインタープリタとは，マシン語しか理解できないコンピュータのために，BASICのコマンドやステートメントの処理の仕方を示した，マシン語でできているプログラムのことです．このようなプログラムがあらかじめメモリに書き込まれており，電源を入れた時点で自動的に実行されるため，BASICが使えるようになります．

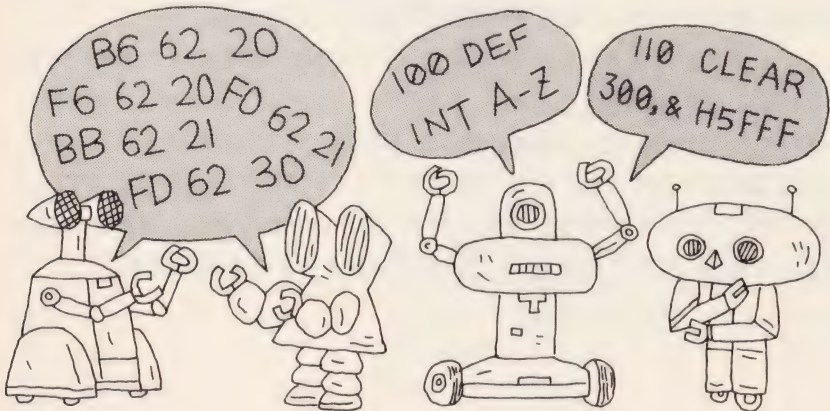
Figure-3.1.1 のオープニング・メッセージが表示され，〝Ready〟が現れている状態が，BASICインタープリタの起動状態です．

```
DISK VERSION
How many disk drives      ?
How many disk files(0-15)?

FUJITSU F-BASIC Version 3.0
Copyright (C) 1981 by FUJITSU/MICROSOFT
25775 Bytes Free

Ready
```

Ready……プロンプト〝Ready〟が表示されてBASICが起動した
その裏ではBASICインタープリタというマシン語のプログラムが実行されている

これはFM-77のオープニングメッセージ．これ以外の機種でもだいたい似たようなメッセージが表示される

Figure-3.1.1　BASICインタープリタが実行されている状態

この状態で私たちはBASICのプログラムを書き，またそのプログラムを実行することができるのですが，コンピュータの内部ではBASICインタープリタが実行され続けているからこそ，正しく処理をしてくれるのです．つまりBASICが動いているように見えるときも，CPUは常にマシン語のプログラムを実行しているのです．

## インタープリタの働き

それではここで，BASICのプログラムが，マシン語で書かれているBASICインタープリタによって実行されていることを確認してみましょう．

まずFigure-3.1.2のBASICのプログラムを入力してください．ただし，BASICインタープリタは機種によって異なりますので，それぞれFigure-3.1.3に示した実行手順に従ってください．

Figure-3.1.2　BASICインタープリタ確認用プログラム

```
10 FOR I=0 TO 9
20  PRINT "I am 6809 CPU."
30 NEXT
```

"I am 6809 CPU."を10回表示する

Figure-3.1.3　BASICプログラムをモニタから実行した様子

```
Ready
MON↵
*G9012↵
I am 6809 CPU.
I am 6809 CPU.
I am 6809 CPU.
I am 6809 CPU.
I am 6809 CPU.
I am 6809 CPU.
I am 6809 CPU.
I am 6809 CPU.
I am 6809 CPU.
I am 6809 CPU.

Ready
```

MON↵……モニタモードにはいる

*G9012↵……BASICインタープリタのRUN命令の処理ルーチンを実行する
＄9012番地はFM-7/77/NEW7などのエントリ・アドレス
LEVEL-3,S1のBモードは＄A5BD番地、FM-8は＄94BE番地を指定する

BASICプログラムが実行される

このBASICのプログラムは"I am 6809 CPU."というメッセージを10回表示するものですが，Figure-3.1.3ではモニタのGコマンドを使ってRUN命令と同じ結果を得ていることに注目してください．実は，Figure-3.1.3で，モニタからGコマンドを使って実行したマシン語のプログラムは，BASICインタープリタの一部で，RUN命令が入力されたときの処理の仕方を示しているプログラムだったのです．つまりBASICでRUN命令を打ち込むと，いつもこのプログラム(Gコマンドで指定したアドレスから始まるRUN命令の処理ルーチン)を実行していたわけです．それを直接モニタから実行させたのですから，同じ結果になって当り前です．

このマシン語のプログラム(BASICインタープリタのプログラム)は，メモリに記憶されているBASICのプログラムを順次読み出しては，各命令に対応するマシン語のプログラムを実行して行きます．Figure-3.1.4は，BASICインタープリタにより，BASICのプログラムが実行されているときの様子を図示したものです．

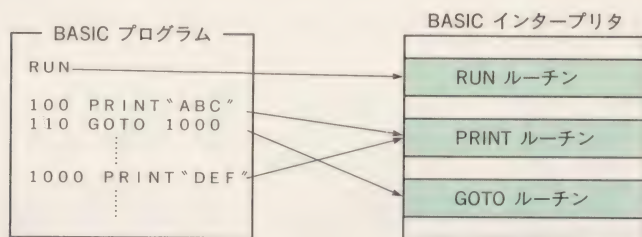



Figure-3.1.4 BASICプログラムの実行とインタープリタの働き

この図から，BASICインタープリタはBASICのプログラムを1行ごとに解読し，自分自身が用意しているマシン語プログラムに置き換えながら，そのマシン語プログラムを実行していることがわかると思います．

さて，説明が少し複雑になりましたのでFigure-3.1.5にコンピュータ，BASICインタープリタ(マシン語)そしてBASICプログラムの関係をまとめておきます．この図ではコンピュータを取り巻く各層が見えていますが，

通常私たちに見えているのは，この表面だけなのです．つまり，電源 ON でコンピュータは BASIC の層に包まれてしまうために，BASIC の表面しかさわれなくなってしまうのです．

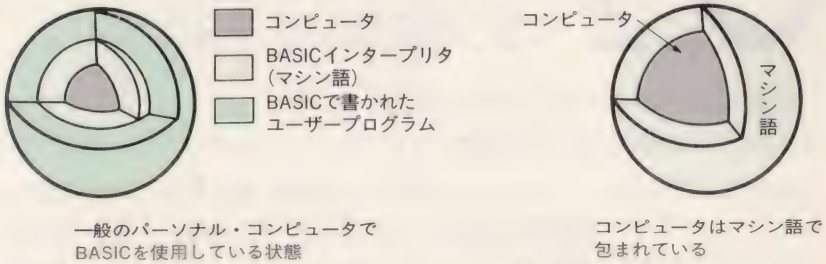
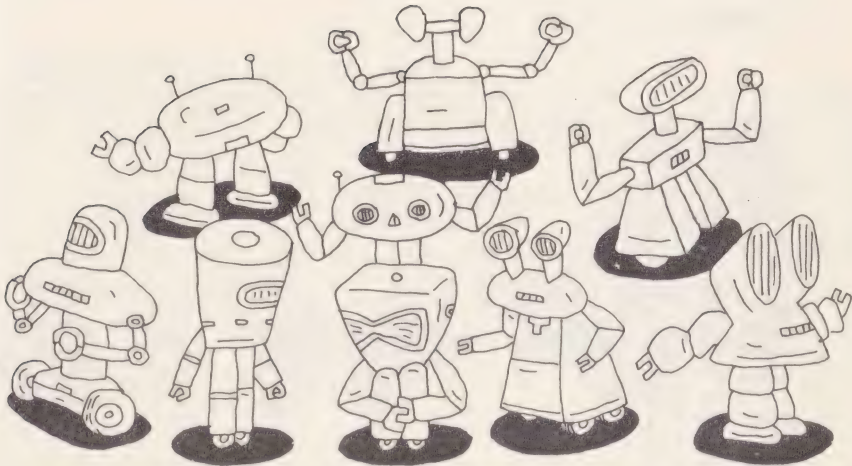



Figure-3.1.5 コンピュータ・システムの概念図

# 3.2 マシン語で命令してみよう

前節ではBASICとマシン語の本質的な違いやコンピュータ・システムにおけるマシン語の位置付けを説明しましたが，このへんでマシン語のプログラムを直接実行させて，コンピュータがマシン語で動作することを確認しておきます．メモリに直接書き込むマシン語のプログラムは，＄００～＄ＦＦで表される16進数の羅列です．これらの数値がコンピュータに対する各種の命令を表したり，アドレス，データとしての数値を表すのです．詳しいことは後の章に譲るとして，ここでは〝コンピュータはマシン語で動く〟ことを体験してください．

まずモニタを起動して，初期設定ができていることを確認してください．次にMコマンドを使って，Figure-3.2.1のとおりにマシン語のプログラムを入力します．その後Dコマンドでマシン語のプログラムが正しく書き込まれたかを確認してください．

Figure-3.2.1　マシン語プログラムの入力と確認

```
*M6000↵ ……モニタのMコマンドでプログラムを入力する
6000 00-7C↵
6001 00-60↵
6002 00-30↵
6003 00-3F↵
6004 00-74↵
6005 00-60↵
6006 00-30↵
6007 00-3F↵
6008 00-B6↵
6009 00-60↵      働きの異なる3つのマシン語プログラムを入力する．全部で18バイト
600A 00-30↵
600B 00-C6↵
600C 00-03↵
600D 00-3D↵
600E 00-F7↵
600F 00-60↵
6010 00-30↵
6011 00-3F↵
6012 00-.↵ ……ピリオドでMコマンドを終了する
```

```
*D6000↵ ………正しく入力されているかDコマンドで確認する

6000 7C 60 30 3F 74 60 30 3F ──入力したプログラム
6008 B6 60 30 C6 03 3D F7 60
6010 30 3F 00 00 00 00 00 00
6018 00 00 00 00 00 00 00 00
6020 00 00 00 00 00 00 00 00
6028 00 00 00 00 00 00 00 00
6030 00 00 00 00 00 00 00 00
6038 00 00 00 00 00 00 00 00
*
```

このプログラムは＄６０３０番地の内容を変える３つのプログラムから成っており，それらは，それぞれ次のような動作を行います．

**＄６０００……＄６０３０番地のメモリの内容に１を足す**
**＄６００４……＄６０３０番地のメモリの内容を1/2にする**
**＄６００８……＄６０３０番地のメモリの内容を３倍にする**

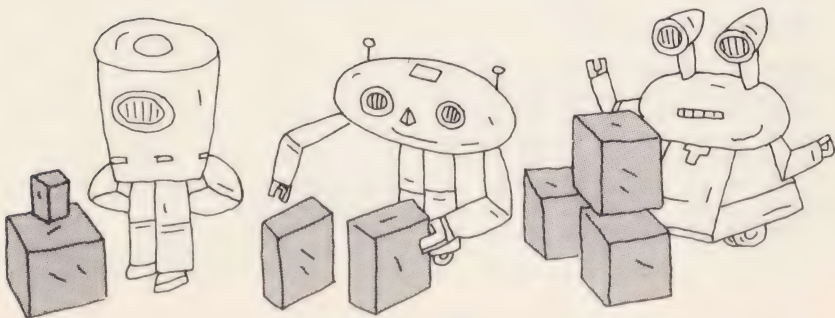
この＄６００４とか＄６００８という数字は，そのアドレスから(Gコマンドにより)実行するという意味で，各動作を行うプログラムの最初のアドレスのことです．このようなアドレスのことを**エントリ・アドレス**といいます．

ではさっそく実行させてみましょう．＄６０３０番地の内容に注意しながら，Figure-3.2.2のように実行してください．

Figure-3.2.2　マシン語プログラムの実行と結果の確認

```
*M6030↲………プログラム実行前に＄６０３０番地に適当なデータを書き込んでおく
6030 00-7F↲………ここでは＄７Ｆ
6031 00-.↲

*G6000↲ ….Gコマンドでプログラム（＄６０３０番地のメモリの内容に１を足すプログラム）
                                                                を実行する
*D6000↲………Dコマンドで実行結果を確認する

6000 7C 60 30 3F 74 60 30 3F
6008 B6 60 30 C6 03 3D F7 60
6010 30 3F 00 00 00 00 00 00
6018 00 00 00 00 00 00 00 00
6020 00 00 00 00 00 00 00 00
6028 00 00 00 00 00 00 00 00     ＄６０３０番地のメモリの内容が＋１された
6030 80 00 00 00 00 00 00 00       ＄７Ｆ＋１＝＄８０
6038 00 00 00 00 00 00 00 00
*G6004↲………＄６０３０番地のメモリの内容を１／２にするプログラムを実行する

*D6000↲………結果を確認する

6000 7C 60 30 3F 74 60 30 3F
6008 B6 60 30 C6 03 3D F7 60
6010 30 3F 00 00 00 00 00 00
6018 00 00 00 00 00 00 00 00
6020 00 00 00 00 00 00 00 00
6028 00 00 00 00 00 00 00 00
6030 40 00 00 00 00 00 00 00     ＄６０３０番地のメモリの内容が1/2になった
6038 00 00 00 00 00 00 00 00       ＄８０／２＝＄４０
*G6008↲………＄６０３０番地のメモリの内容を３倍にするプログラムを実行する

*D6000↲………結果を確認する

6000 7C 60 30 3F 74 60 30 3F
6008 B6 60 30 C6 03 3D F7 60
6010 30 3F 00 00 00 00 00 00
6018 00 00 00 00 00 00 00 00
6020 00 00 00 00 00 00 00 00
6028 00 00 00 00 00 00 00 00     ＄６０３０番地のメモリの内容が3倍になった
6030 C0 00 00 00 00 00 00 00       ＄４０＊３＝＄Ｃ０
6038 00 00 00 00 00 00 00 00
*
```

１つ実行するたびに＄６０３０番地の内容が変化しています．やっていることは単純ですが，これはまぎれもなくマシン語を実行した結果なのです．

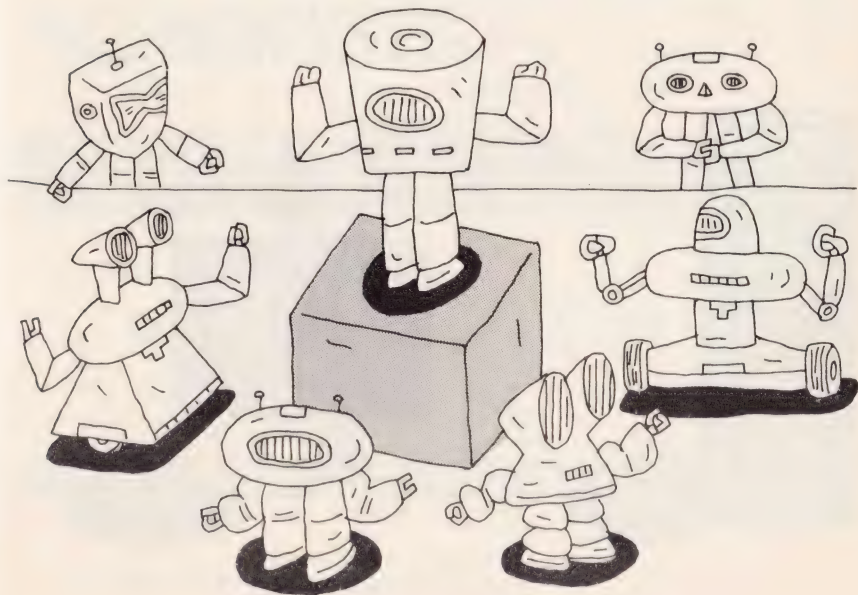
# 4 コンピュータ・システムの構造

●マシン語を理解するためには，コンピュータ・システムのアーキテクチャを理解しなければなりません．マシン語のプログラムとハードウェアとは密接な関係にあり，その両者を把握できてこそコンピュータの本質に迫れるのです．

本章ではコンピュータ・システムのメインであるCPU，メモリ，I／Oについて，その構造と関係を順に解説していきます．

# 4i CPUの働き

CPU とは Central（セントラル） Processing（プロセッシング） Unit（ユニット）(中央処理装置)の略で，コンピュータ・システムの心臓部です．コンピュータ自身といった方がよいかもしれません．

本書で扱う 6809 は，米モトローラ社の 6800 シリーズの最上位 CPU として発表されたもので，究極の 8 ビット CPU とまでいわれており，その豊富な命令セットや処理速度などの点から，現在ある 8 ビット CPU の中では最も強力なものです．そのアーキテクチャ*1は，同社の一貫した設計思想により非常に洗練されたものになっています．

*1 コンピュータのソフトウェア，ハードウェアにわたるシステムの構造

さて Figure-4.1.1 は，CPU を中心としたコンピュータ・システムの構成を示したものです．この図を見てもわかるように，CPU，メモリ，I／O，バスといったいくつかの基本的なブロックの組合せによって構成されています．CPU からは 16 本の〝アドレスバス〞と呼ばれるアドレス線と，8 本の〝データバス〞と呼ばれるデータ線が出ています．これらはメモリと CPU をつなぐためのもので，これを使って CPU とメモリとの間でデータのやり取りが行われるのです．つまり，アドレスが 16 ビットで表され，データが一度に 8 ビットまとめて処理されるというのは，実はこの 2 つの線の本数のことなのです．

CPU は，これらのバスを利用してメモリからマシン語の命令を読み出し，命令の解析および実行を行います．CPU の実行とはデータの転送や演算，判断，分岐などの動作をいい，これはそのままコンピュータの動作ということができます．CPU 以外のものは，ただ CPU の働きに応じて動いているにすぎないのです．

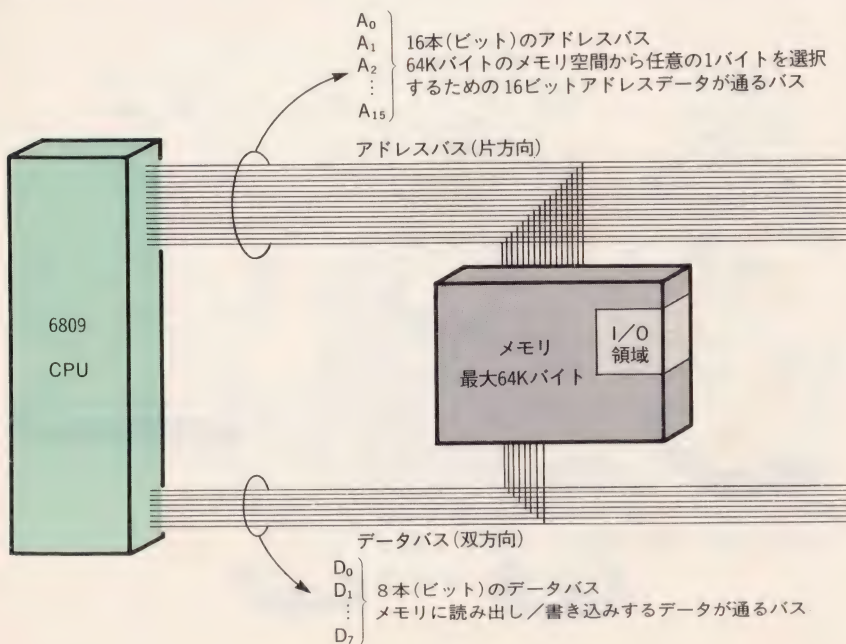
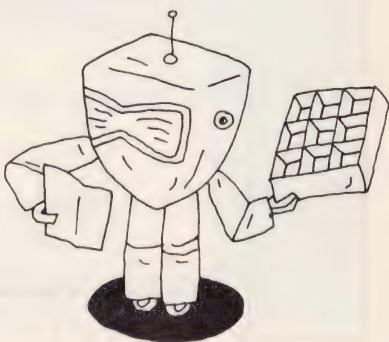
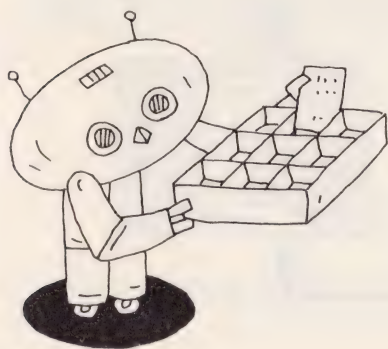
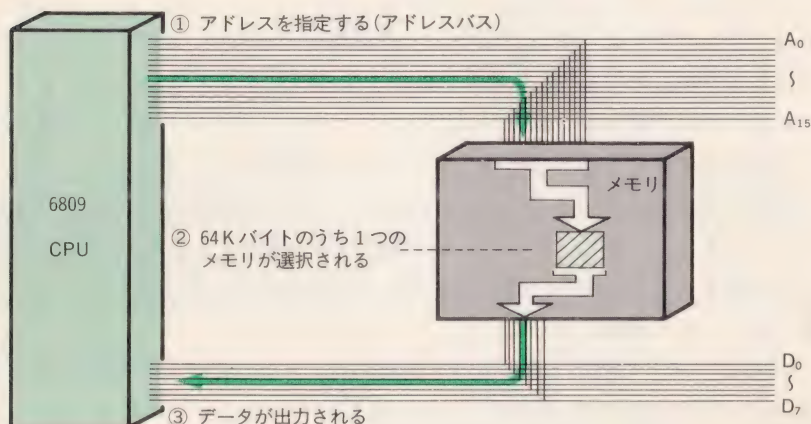



Figure-4.1.1　コンピュータ・システムの構成

# 4.2 メモリの働き

コンピュータ・システムのなかで一番重要なのはCPUということになるのでしょうが，そのCPUもメモリがなくては何もできません．CPUとメモリとは非常に密接な関係にあり，CPUのアーキテクチャはメモリとの読み出し／書き込み動作を中心に考えられているのが普通です．現にいままでいくつかプログラムを実行させてきましたが，プログラムはすべてメモリに書き込んでいました．また，プログラムの実行結果の確認にしても，CPUが処理した結果をメモリに残し，後からメモリの内容を表示して，プログラムが実行されたことを確認しました．

CPUがメモリの内容を読み出す，あるいは，メモリにデータを書き込む場合には，CPUはまず16本のアドレスバスにアドレスを出力します．これによって64Kバイトのメモリの中からたった1つのメモリが選び出され，メモリからその内容がデータバスへ出力されたり(読み出し)，CPUがデータバスに出力したデータをそのメモリにしまいます(書き込み)．

これらの様子を示したものが **Figure-4.2.1** です．この図には示されてはいませんが，実際にはアドレスバス，データバスのほかに，読み出し，書き込みの区別や，それらの動作のタイミングをとるためのコントロールバスと呼ばれる制御線も何本か CPU から出力されます．

(1) 読み出し

(2) 書き込み

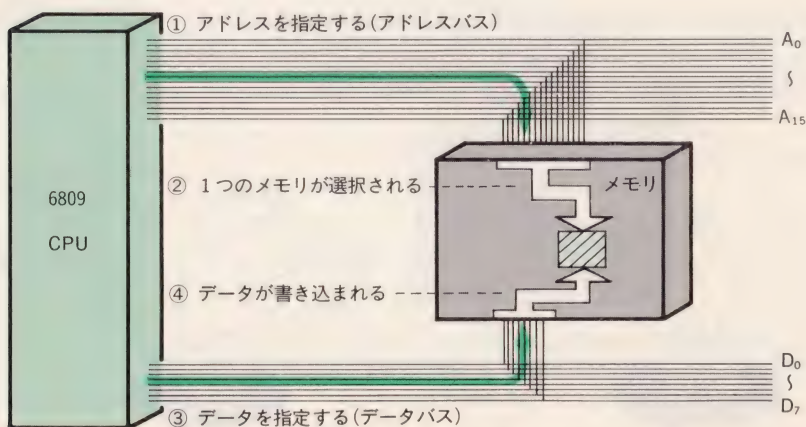



Figure-4.2.1　メモリの読み出し／書き込みの様子

## RAMとROM

メモリには大きく分けて読み出しも書き込みもできるRAM(Random Access Memory)と，読み出ししかできないROM(Read Only Memory)の２種類があります.RAMは自由に読み書きができないと困るユーザー用のプログラムエリアやワークエリアに使われており，すでにみなさんがデータやプログラムを入力した＄６０００番台(＄６０００～＄６ＦＦＦ)のメモリもRAMになっています. 一方，ROMは書き込むことはできませんが，コンピュータの電源を切ってもその内容が消えないため，電源ONと同時に起動するBASICインタープリタなどの格納に使われています.

ROMに書き込みができないことを除いては，RAMもROMもCPUから見ればまったく同じメモリであり，実際には64Kバイトのメモリ空間にRAMとROMを混在させています. それらの具体的なアドレスにについては，巻末付録に各機種のメモリマップを掲載しておきましたので，それを参照してください.また，どの機種も＄Ｅ０００番台(＄Ｅ０００～＄ＥＦＦＦ)はROMになっていますから，試しにモニタを使ってそこに読み書きをしてみればROMの性質が理解できると思います.

# 4.3 I/Oの働き

コンピュータの入出力装置のことをI/O(Input Output System)と呼んでいます．たとえCPUがプログラムを実行しても，その結果をスクリーンやプリンタ，ディスクなどに出力しなければ，私たちはプログラムの実行結果を確認することはできません．またキーボードがなければ，CPUに命令を与えることもできません．このように，I/Oとは人間とのコミュニケーションをとるために必要なものです．では実際にどのようにCPUと外部の入出力装置とでデータのやり取りを行うのか，その仕組みを説明しましょう．

## “I/O領域”

メモリの64 Kバイトのアドレス空間の一部には，CPUと入出力装置とのデータのやり取りのために割り当てられているメモリがあります．この特殊な領域を**I/O領域**と呼び，CPUと入出力装置はこのメモリを経由して様々なデータや信号(ステータス情報)をやり取りするのです．

例えば，あるアドレスのメモリをこのI/O領域として割り当て，そのメモリにキーボードから出力されるアスキーコードを書き込んでやれば，CPUがそれを読み出すことによって，入力された文字を知ることできるのです．

I/O領域のメモリは，プリンタやディスクなどの入出力装置のデータや信号をデータバスに出力するための接点になっているので，それぞれの入出力装置ごとにアドレスを割り当てておけば，CPUは自由に外部機器とコミュニケーションをとることができるのです(Figure-4.3.1)．

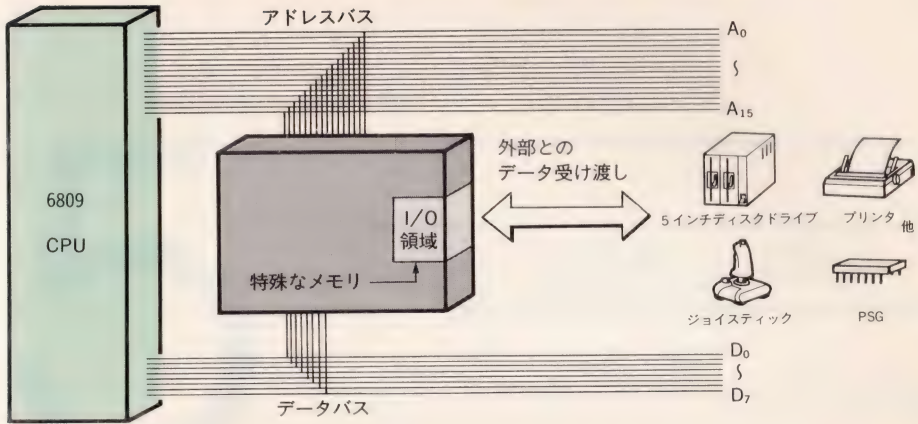



Figure-4.3.1 I／O領域と外部とのデータの受け渡し

普通のメモリとI／O領域のメモリとは，構造も働きもまったく異なるのですが，いずれも同じ64Kバイトのアドレス空間の一部であり，CPUから見ればまったく同様に扱うことができる存在です．Figure-4.3.2に示したように，メモリの最もシンプルなモデルとして64KバイトをRAMとROMとI／O領域がそれぞれ場所を分け合って並んでいる姿を想像してください．

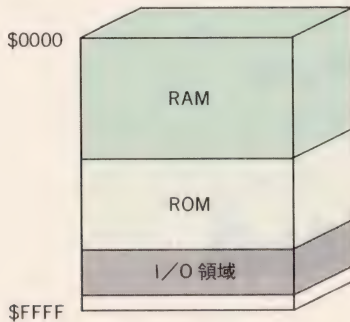
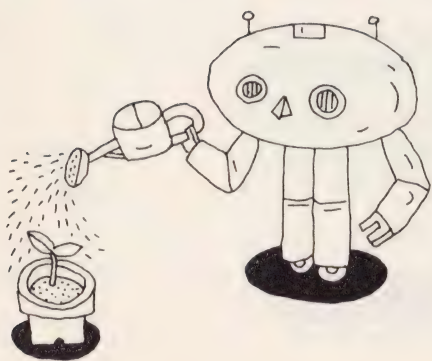



Figure-4.3.2 アドレス空間におけるRAM，ROM，I／O領域

# 5

## コンピュータの中心　CPU

●いよいよコンピュータの核心部分の説明にはいります．

コンピュータの動作はすべてCPUを中心に行われているため，CPUの働きやその内部構造，とりわけレジスタの働きについては十分な知識が必要です．

本章ではこのCPU内部のレジスタを１つ１つ取り上げ，その機能について紹介します．また，CPUがどのようにメモリ上のプログラムを実行しているのかについては，実際に動作しているプログラムを例にあげて，実行されるプロセスを概観します．

# 5.1 CPUの内部

マシン語プログラムで行う比較，演算といった様々な操作をCPU内部で行うために，メモリに置かれたデータは，いったんCPU内部の記憶装置に取り込まなければなりません．例えば〝$6000番地のメモリの内容を3倍にする〟には，まず$6000番地のメモリの内容をCPUの内部に取り込み，CPU内部で3倍するという演算を行い，その結果をメモリに戻すのです．この CPU 内部の記憶装置を**レジスタ**と呼び，Figure-5.1.1に示したように6809の内部には，全部で9個のレジスタがあります．レジスタは数値を保存するという面ではRAMなどと同じメモリですが，その目的は少々異なります．

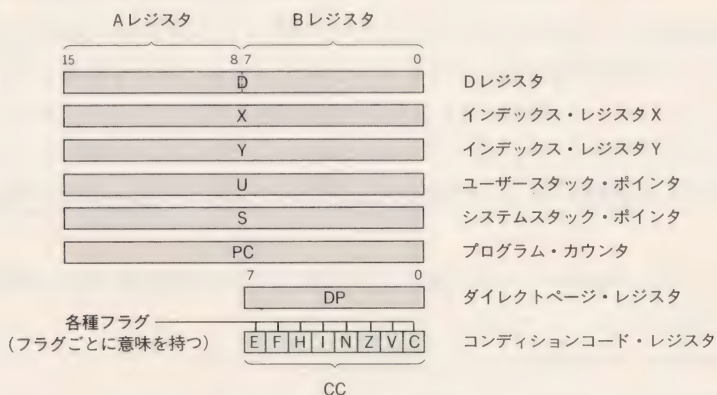



Figure-5.1.1　6809 CPUのレジスタ

レジスタの役割は，演算をしたり，演算結果の状態(繰り上がりやゼロ)を表したり，実行すべきプログラムのアドレスを示したり…と様々ですが，これらの機能は各レジスタに分担させてあります．

マシン語のプログラミングとは，レジスタとメモリの間で転送や演算を行ったり判断をすることによって，必要な結果を得ることです．そのため，レジスタの種類とその機能をよく理解しておかなければなりません．しかし，各レジスタの機能を十分理解するにはどうしても7章以降のマシン語の命令と合せて理解しなければならないので，ここではそれぞれのレジスタのプロフィールを紹介しておくにとどめます．

## “アキュムレータ(Accumulator) A, B, D”

**アキュムレータ**と呼ばれる〝A〟，〝B〟2つのレジスタは，ともに8ビットのレジスタです．つまり，どちらも1バイトのデータを扱うことができます．またAレジスタを上位バイト，Bレジスタを下位バイトとして2バイト，すなわち16ビットのアキュムレータ〝D〟として使用することもでき，命令はA，B，Dそれぞれのレジスタについて用意されています．

アキュムレータは，算術演算(+, −, * など)や論理演算(AND, OR など)などに使用でき，そのほかにも何かにつけて使用される最も使用頻度の高いレジスタです．命令の数もほかのレジスタよりずっと多くなっています(Dレジスタは A，Bレジスタに比べるとやや少ない)．

## “コンディションコード・レジスタ(Condition Code register) CC”

**コンディションコード・レジスタ**とは，その名のとおり演算結果やCPUの動作の状態を表すレジスタのことです．これも8ビットのレジスタですが，その内容は8ビットの数値としては意味を持っていません．1ビットで1つの意味を持つような情報のことを特に**フラグ**(Flag)といいますが，このレジスタは Figure-5.1.1 に示したように，それぞれ違った意味を持つ8つのフラグの集まりとして働くレジスタなのです．

このレジスタは通常はプログラムで積極的に読み出したり書き込んだりはしません．いろいろな命令を実行した結果によってCPU内部で自動的に変

更されるので，プログラムではそれを利用して条件判断などの材料にしています．ですから，命令実行後の各フラグのセット／リセットのされ方(ビットが1になっているか0になっているか)がわからないと，マシン語のプログラムがどのように実行されるのかがわからなくなってしまいます．

フラグについては7章以降で詳しく解説を行いますが，これらの8つのフラグすべてを詳細に説明することは〝はじめて読む〟という主旨からも不適当であると思われるので，マシン語を理解するために特に重要なC, Z, Nの3つのフラグにしぼって解説していきます．

## “ インデックス・レジスタ(Index register) X, Y ”

**インデックス・レジスタ**とはアドレスを指し示すためのレジスタのことで，そのため〝X〟も〝Y〟も16ビットの長さを持っています．そもそもインデックスとは〝指針〟とか〝指標〟といった意味があり，その名前はこれらのレジスタの役割をよく説明しています．

例えばXレジスタの内容(16ビット)をアドレスとみなしてそのアドレスの内容を参照したり，そのアドレスへプログラムの実行を移したりするのです．詳細は7章と14章で解説しています．

X, Yは名前こそインデックス・レジスタですが，データの保存はもちろん，加減算もできるので汎用の16ビット・レジスタとしても使えます．

## “ スタック・ポインタ(Stack pointer) S, U ”

**スタック・ポインタ**とはスタックを管理するレジスタです．スタックの説明はここでは控えますが(12章参照)，このレジスタもXレジスタやYレジスタと同じくアドレスを指すために16ビットになっています．SレジスタやUレジスタからスタック・ポインタとしての機能を差し引いて考えれば，X, Yレジスタとまったく同じ機能になります．

S，Uをそれぞれ**システムスタック・ポインタ**(System stack pointer)，**ユ**

**ーザースタック・ポインタ**(User stack pointer)といい，その働きは若干異なります．平たくいえばSレジスタはシステム(CPU)が管理するレジスタであり，Uレジスタはユーザー(プログラマ)が管理するレジスタです．

## ダイレクトページ・レジスタ(Direct Page register) DP

**ダイレクトページ・レジスタ**はアドレスの指定を軽減するための8ビットのレジスタです．64Kバイトのメモリ空間のなかから1つのアドレスを指定するには16ビット(2バイト)必要ですが，もし参照したいいくつかのアドレスの上位8ビットが等しければ(例えば＄7200，＄7201，＄7202のように)，毎回2バイトで表すのは不経済というものです．こういう場合，初めに上位バイトを指定しておいて，それ以降は下位8ビットのみでアドレスの指定ができれば非常に便利です．

このような目的のために用意されたのがDPレジスタで，上位バイトをDPレジスタにセットすれば，そのDPレジスタで表された256バイトの空間に限り，1バイトで参照することができます．

実際のプログラムでも，よく使うアドレスは限られているので，そこにDPレジスタをセットしておけば効率よくアドレスを参照することができるのです．BASICインタープリタのワークエリアなどにもこのDPレジスタが利用されています．

## プログラム・カウンタ(Program Counter) PC

実行すべき命令が格納されているアドレスを指すレジスタを，**プログラム・カウンタ**といいます．このレジスタは命令の実行をコントロールするもので，レジスタのなかでは最も重要なものの1つです．アドレスを指し示すため16ビットの長さを持っていますが，機能的にはカウンタというよりはポインタといった方がその役割をよく表しています．次節でプログラムの実行メカニズムを追いながら，もう少し詳しく解説します．

# 5.2 プログラム実行のメカニズム

現在利用されているほとんどのコンピュータの設計思想は，フォン・ノイマンの提案したものを基礎としています．これらフォン・ノイマン型と呼ばれるコンピュータの基本原理は，メモリ上にプログラムを置き，それを順次CPUに取り込み，解析／実行をする方式(ストアード・プログラム・シーケンシャル・コントロール方式)であり，6809もこの方式にのっとったCPUです．

本節では，CPUのプログラム実行のメカニズムを，3章のFigure-3.2.1のプログラムを例に解説します．これは，すでに実行したとおり3つのプログラムから構成されていますが，Figure-5.2.1に示すように，＄６００８番地から始まる10バイトのプログラムを使います．なお，＄６０３０番地には初め＄０２がはいっているものとします．

| アドレス | 6008 | 6009 | 600A | 600B | 600C | 600D | 600E | 600F | 6010 | 6011 |
|---|---|---|---|---|---|---|---|---|---|---|
| 内　容 | B6 | 60 | 30 | C6 | 03 | 3D | F7 | 60 | 30 | 3F |

Figure-5.2.1　メモリ上のプログラムの内容

メモリ上の10バイトのマシン語のプログラムは，命令を表す部分と命令に与えられるデータの部分が混在しています．図のなかで色の付いている部分が命令を表し，その他の部分がデータです．これらの16進数を見ただけでは，いまはまだ何のことかわからないと思いますが，ここで重要なのは，プログラムのスタート・アドレスからメモリに記憶されているプログラムが1バイトずつ読み出され，CPUによってどのように解析／実行されていくかということです．

これら一連の動作を，図とともに解説したのが Figure-5.2.2 です．注意して見て欲しいのは，CPU の実行プロセスとプログラム・カウンタの値（プログラムを読み出すアドレス）によってコントロールされるプログラムの実行順序です．この図からプログラムの実行とはどういうことなのか，そのプロセスの概要がつかめるのではないかと思います．

| アドレス | 内容 | CPUの動作 | PC プログラム カウンタ | A レジスタ | B レジスタ | $6030番地の内容 |
|---|---|---|---|---|---|---|
| $6008 | B6 | 命令コードを読み出し，それを解析する<br>[$B6 :〝次の２バイトを読んで，その値（$6030）の指すアドレスの内容をＡレジスタに入れよ〟] | 6009 | ? ? | ? ? | 02 |
| $6009 | 60 | この２バイト１組で16ビットデータ（$6030）を表す | 600A | ? ? | ? ? | 02 |
| $600A | 30 | | 600B | 02<br>（$6030番地の内容がＡレジスタにはいった） | ? ? | 02 |
| $600B | C6 | 命令コードを読み出し，それを解析する<br>[$C6 :〝次の1バイトのデータ（$03）をＢレジスタに入れよ〟] | 600C | 02 | ? ? | 02 |
| $600C | 03 | Ｂレジスタに入れる値 | 600D | 02 | 03<br>（命令コードの次のアドレスの内容がＢレジスタにはいった） | 02 |
| $600D | 3D | 命令コードを読み出し，それを解析する<br>[$3D :〝ＡレジスタとＢレジスタを掛けて，その結果をＤレジスタに入れよ〟] | 600E | 00 | 06 | 02 |
| | | | | （Ａレジスタ）×（Ｂレジスタ）<br>$02×$03＝$0006 | | |
| $600E | F7 | 命令コードを読み出し，それを解析する<br>[$F7 :〝次の2バイトを読んで，その値（$6030）の指すアドレスにＢレジスタの内容をしまえ〟] | 600F | 00 | 06 | 02 |
| $600F | 60 | この２バイト１組で16ビットデータ（$6030）を表す | 6010 | 00 | 06 | 02 |
| $6010 | 30 | | 6011 | 00 | 06 | 06<br>（Ｂレジスタの内容（$06）が$6030番地にはいった） |
| $6011 | 3F | 命令コードを読み出し，それを解析する<br>[$3F :〝モニタへ戻れ〟 *1] | | 00 | 06 | 06 |

※重要：プログラムを１バイト読むとすぐにPCがインクリメント（＋1）されることに注意！
たとえば$6008番地のデータ（この例の場合は命令）を読んだ瞬間PCは$6009になっている

＊1　正確には〝$FFFA〜$FFFBに書かれているアドレスに分岐せよ〟

Figure-5.2.2　プログラム実行のプロセス

図のようなプロセスにより，＄６００８番地から始まる 10 バイトのプログラムの実行は終わり，その結果＄６０３０番地の内容が 3 倍され，モニタに戻ってプロンプトが表示されます．かなり複雑なプロセスですが，ここで説明したことを，CPU はたったの 0.00003 秒程度で処理してしまいます．

# 6 アセンブリ言語とアドレッシングモード

●前章まではコンピュータのアーキテクチャに関する解説を中心に行ってきましたが，本章から，いよいよマシン語そのものの解説にはいります．

まず最初に，マシン語の記号や言葉を説明します．マシン語のプログラムを書くには，ニーモニックと呼ばれる記号を使ったアセンブリ言語で記述するのですが，ここでは16進数の羅列であるマシン語とアセンブリ言語が，どのような関係があるかについて述べます．

さらに，マシン語の重要な概念であるアドレッシングモードについて説明します．アドレッシングはCPUのアーキテクチャと密接な関係があり，マシン語を理解する上で欠かせない概念であるとともに，効率のよいマシン語プログラムを書くためには，必ず理解していなければならない事項です．

# 6・1 マシン語とニーモニック

いままで実行してきたマシン語プログラムは，すべて 16 進数の羅列で表現されていました．下に示したのは 3.2 節で実行したプログラムですが，このただの 16 進数の並びは，前章で説明した実際の CPU の動作とは，はっきりいって何の脈絡も見い出せません．6809 のマシン語に精通している人のなかには，これだけを見て何の動作をするプログラムなのか理解してしまう人もいるようですが，そんな人ですらもっと大きなプログラムになれば全体を把握するのは難しくなります．CPU にとって理解しやすいマシン語も，人間の思考方法にはまったく合わないものなのです．

```
6008      B6  60  30  C6  03  3D  F7  60
6010      30  3F
```

このような理由から，実際にマシン語を扱うときは人間の言葉に近く，なおかつ簡潔な方法でマシン語を扱える**アセンブリ言語**(Assembly language)というものを使います．下に示したのが，上のプログラムをアセンブリ言語で書いたものですので，まず両者をよく見比べてください．

```
ニーモニック
 LDA    >$6030
 LDB    #3
 MUL
 STB    >$6030
 SWI
```

アセンブリ言語は＄B6とか＄3Dのような命令を直接書くのではなく，それらの命令と1対1に対応した**ニーモニック**(Mnemonic)を使用します．ニーモニックとは，各命令の意味がわかりやすいように考えられた数文字のアルファベットから成る一種の英単語のことです．さきの例では，LDA, MUL, STBなどがニーモニックです．それぞれ，ロードA(LoaD A)，マルチプライ(MULtiply)，ストアB(STore B)というように読みます．LD, MUL, STは各動作を意味する英語の一部を取ったもので，その後ろに続くAやBは前章で紹介したレジスタの名前です．初めはやはりとっつきにくいかもしれませんが，ちょっと慣れればニーモニックを見てすぐにその意味を知ることができるようになります．

## “プログラム開発の手順”

マシン語プログラムの開発は，まずニーモニックを使ったアセンブリ言語で書きます．このアセンブリ言語で書かれたプログラムは，マシン語プログラムのもととなるプログラムなので，**ソース・プログラム**(Source Program)と呼ばれ，この時点でプログラムに誤りがないかを十分に調べます．そしてOKとなってから，マシン語に変換します．このマシン語に変換されたプログラムを**オブジェクト・プログラム**(Object Program)と呼びます．

ニーモニックからマシン語に変換する作業を**アセンブル**といい，アセンブルは普通**アセンブラ**と呼ばれるプログラムによって処理されます．しかしこれは，すでにマシン語を知っている人がプログラムを開発するときの話であり，マシン語を学習しようとしている人の話ではありません．私たちが行うアセンブルは，巻末に掲載した命令表を使って，1つ1つのニーモニックをマシン語に変換するという方法をとります(このことを俗に**ハンドアセンブル**という)．この方法は，数Kバイトにも及ぶ大きなものになると膨大な時間がかかりますが，本書で扱うプログラムは小さいものばかりなので，マシン語の理解を深めるためにもハンドアセンブルしてください．なお，ハンドアセンブルの方法は，7章の転送命令で説明します．

# 6.2 アドレッシングモード (Addressing mode)

前節で参照したソース・プログラムを，ここでもう一度取り上げます.

```
LDA    >$6030
LDB    #3
MUL
STB    >$6030
SWI
```

前節でも述べたように，LDA，MUL，STB などのニーモニックは，それぞれ固有のマシン語の動作を意味しています. LDA は〝A レジスタにデータを転送する〞，MUL は〝掛け算をする〞，STB は〝B レジスタの内容をメモリに転送する〞といった意味を持っていますが，これらの示している意味は，「どこから」あるいは「何を」という情報が欠けているのです. LDA では A レジスタに〝どこからデータを転送する〞のかが示されていません. また，STB は B レジスタの内容を〝メモリのどこへ転送する〞のかがわかりません. これら命令の目的語となる部分は，ニーモニックには示されておらず，ニーモニックに続く記号や数値によって示されます.

このようにニーモニックの動作の対象となるデータを，どこからあるいはどこへの部分を指定することを**アドレッシング**と呼んでいます. 6809 には，Figure-6.2.1 に示した 6 種類のアドレッシングモードがありますが，これらのアドレッシングモードを理解することによって，マシン語命令の全体像を把握することができるようになり，効率のよいプログラムが書けるようになります.

| アドレッシングモード | 各モードの記号 | アドレッシングの指定方法 |
|---|---|---|
| エクステンド (extend) | > | 命令が参照するオペランド*1を16ビットのアドレスで直接指定するモード．64Kバイトのメモリ上のすべてのアドレスを直接参照できる．エクステンド・アドレッシングを指定するためにはオペランドの前に〝>〞を付ける |
| ダイレクト (direct) | < | 命令が参照する16ビットのアドレスをダイレクトページ・レジスタ（DP）を上位バイトとして使い，下位バイトのみオペランドとして指定するモード．256バイトまでのデータをアクセスする場合にはエクステンドより高速で短いプログラムの作成が可能である．ダイレクト・アドレッシングを指定するにはオペランドの前に〝<〞を付ける |
| イミディエイト (immediate) | # | 命令が参照する値に定数を使いたい場合に，直接オペランドに書いた値を参照するためのモード．イミディエイト・アドレッシングを指定するためにはオペランドの前に〝#〞を付ける |
| インデックス (index) | , | 直接アドレスやデータをオペランドに書くのではなく，インデックス・レジスタ（X,Y,U,S）等を使用してアドレスの指定をするモード．実行時でなければ参照するアドレスが決まらなかったりスタック上の値を扱う場合に使用する．このアドレッシングのバリエーションには24種ある．その詳細については14章を参照 |
| インヘレント (inherent) | なし | データの指定がいらないか，命令のなかで操作する先が示してあるか暗黙に決められているため，オペランドを指定しないモード |
| リラティブ (relative) | なし | アドレスを，プログラム・カウンタ（PC）すなわち次の命令のアドレスからの相対位置により指定するモード，詳細については13章を参照 |

＊1 命令の動作の対象となるデータ．80ページ参照

Figure-6.2.1　6809のアドレッシングモード

具体例をあげて，もう少し詳しく説明しておきましょう．例えば，次の例を見てください．

LDA　>＄6030……Aレジスタに＄6030番地の内容を転送する(エクステンド)

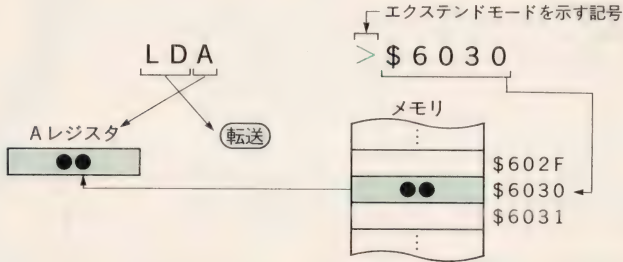



Figure-6.2.2 "LDA >$6030"の動作

これは，LD(転送する)，A(Aレジスタに)，">＄6030"(＄6030番地の内容を)という意味になり，">"が任意のアドレスのメモリからデータを持ってくるという意味を持っています．

また次のような場合，

LDB　#3…………Bレジスタに3という値そのものを転送する(イミディエイト)

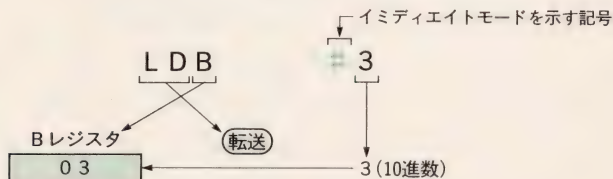



Figure-6.2.3 "LDB #3"の動作

LD(転送する)，B(Bレジスタに)，"#3"(3という値そのものを)という意味になり，"#"が1バイトあるいは2バイトの数値データそのものを示します．

“＞”や“#”はアドレッシングモードを区別するための記号であり，これらを指定せず，ただ3とか＄６０３０と書いても，その数字がデータの値そのものなのか，データのあるアドレスを表すのかわからないのです．

MULやSWIにはアドレッシングモードを区別する記号がありませんが，これらの命令は，すでにその動作が確定しているために，あらためてデータを指定する必要がないのです．例えばMUL命令が実行された場合には，常にAレジスタとBレジスタの積がDレジスタにセットされるようになっています．

このように，命令に与えられるデータの値やデータのあるアドレスの示し方はいく通りもありますが，これらのアドレッシングモードの詳しい説明は，今後，関係する命令のところでそのつど解説することにします．

いよいよ次章から実習にはいりますが，まず，コンピュータの基本的な動作にかかわる一般的な基本命令を正しく理解することです．リアルタイムで動くゲーム・プログラムも，数多くの基本的な命令の組合せでできているのであり，決して特別な操作を行っているわけではありません．

6809は優れた命令を数多く持っており，様々な場面において柔軟に対応できるように設計されています．その数は1464にものぼりますが，これはレジスタやアドレッシングモードの違いによって細かく分類されているためであり，命令の種類(LD，STなど)はそれほど多くはありません．

本書ではこれらの命令を常に体系的に解説していますので，順序どおり読み進んでいけば知らないうちにたくさんの命令があやつれるようになっているでしょう．また，すべての命令を覚えてからでないとプログラムを作れないわけでもありません．1つのプログラムの9割以上は，ほんの10種類ぐらいの命令で占められているので，ある程度のプログラムなら実習の初めの段階から自力で書くことも可能です．大切なことは，ひたすら命令を覚えることではなく，頭の中を整理しながら体系付けて理解することであり，さらに理解したことを実践して自分の力で納得することなのです．

# 7

# 転送命令
# (レジスタ↔メモリ)

●マシン語でいろいろな処理をするときの最初の課題は，メモリとレジスタの間で自由にデータの転送をすることです．

データはもともとメモリ上にあり，それらを十分に活用するには，まずレジスタに転送することが必要です．また，レジスタ上で処理されたデータは，再度メモリに転送しなければ処理結果を使うことができません．

本章では，すべてのマシン語のプログラムの基礎となる，この転送という動作について解説します．その内容は，本章以降の章で必要になることばかりですので，疑問点を残すことのないようにしてください．

# 7.1 LD, ST命令

マシン語の命令の中で最も基本的で,最もよく使われるのが転送命令です.
6809のデータ転送には,

**レジスタ←→メモリ間の転送**
**レジスタ←→レジスタ間の転送**

の2種類がありますが(メモリ←→メモリ間の転送はない), 本章では前者のみを解説します.

レジスタ←→メモリ間の転送はデータの移動の方向により,

**メモリ→レジスタ……ロード**
**レジスタ→メモリ……ストア**

と呼び分けており, ニーモニックはLD(ロード), ST(ストア)と書きます. "転送" というと, あるところから別の場所へデータが移されてしまうような印象を受けますが, 転送元のデータもそのまま保存されるので, "コピー" といった方がこれらの命令の動作を的確に表現していると思います. Figure-7.1.1はLD, ST命令の動作を図解したものです.

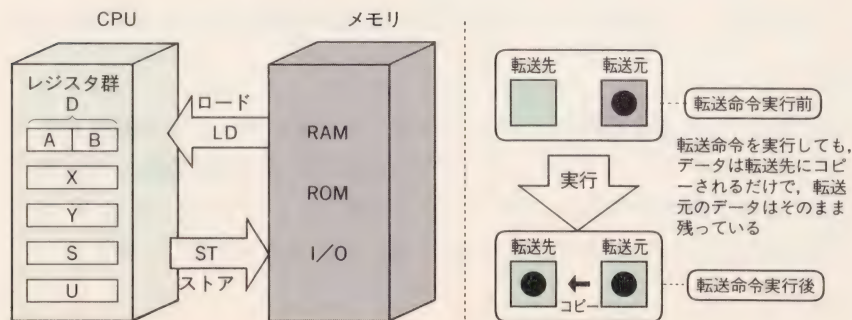



Figure-7.1.1 LD, ST命令の動作

# “LD”

LD(LoaD)命令は，レジスタ A，B，D，X，Y，U，S に対してデータを転送します(CC，DP，PC にはできない)．データの転送元の示し方にはエクステンド，ダイレクト，イミディエイト，インデックスの各アドレッシングモードが使えます．命令は各レジスタと各アドレッシングモードを組み合せて，次のように書きます．

```
LDB   >$0246…$0246番地の内容をBレジスタへコピーする(エクステンド)
LDA   ,X……………Xレジスタの指すアドレスの内容をAレジスタへコピーする(インデックス)
LDD   #$8000…$8000という値をDレジスタへコピーする(イミディエイト)
LDX   <$FE………$00FE番地の内容をXレジスタへコピーする(ダイレクト).ただし,DPレジスタが$00のとき
```

このように，任意のレジスタに任意のアドレッシングモードを使ってデータをコピーすることができます．アドレッシングモードについては，特にイミディエイトとエクステンドの違いについて，Figure-6.2.1 を参照しながら正しく理解してください．

ここで1つ説明しておかなくてはならないことがあります．それは，

```
LDX   >$3081…$3081番地の内容をXレジスタへロードする
```

のようにアドレスを参照して16ビットのレジスタにロードする場合です．“>$3081”というアドレッシングの指定は“$3081番地のメモリの内容”という意味ですから，そのデータは当然8ビットなのに，X は16ビットのレジスタということは，長さが食い違っているため矛盾して見えます．

しかしこれは矛盾しているわけではなく，〝$ 3 0 8 1 番地の内容〟とは〝$ 3 0 8 1 番地の内容を上位バイト，$ 3 0 8 2 番地の内容を下位バイト，とした 16 ビットの値〟の意味なのです．Figure-7.1.2 を見てください．16 ビットのレジスタにアドレスを参照してデータを転送する場合は，参照されたアドレスとその次のアドレスが，転送元となる 16 ビットデータになり，いつもこのような表現をします．なお，このことは LD 命令以外の命令に対しても共通です．

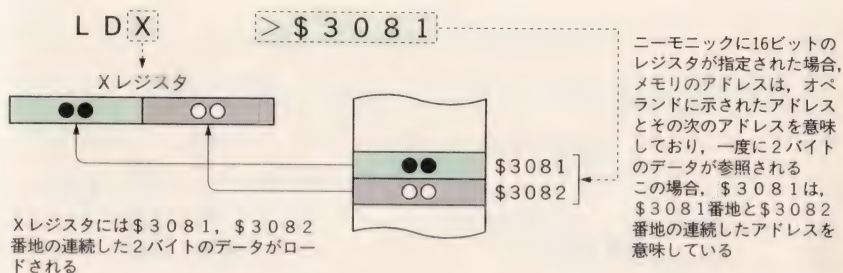



Figure-7.1.2　16ビットレジスタを指定した場合にアクセスされるアドレス

次に，これらの命令がマシン語では実際にどうなるのか，その変換方法(ハンドアセンブル)について解説しましょう．

## ハンドアセンブルの実際

Figure-7.1.3 にロード命令の部分だけ抜き出した命令表*1を掲載しました．この表の見方は，データをロードするレジスタを縦の列から選び，アドレッシングモードを横の列から選んで，その交わる部分を読めば対応する 16 進数で表されたマシン語の命令(これをオペコード：OPeration code(オペレーション コード) という．以下 OP(オペ) コードと略)になります．

*1 APPENDIXの命令表参照

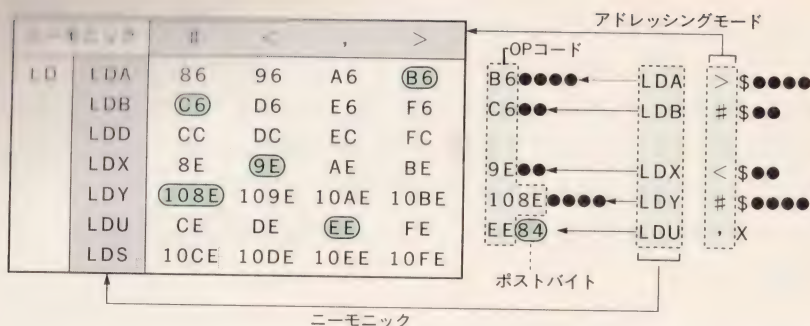



Figure-7.1.3 ハンドアセンブルの方法

このように，表から得られたOPコードに続けてアドレスやデータ(これを**オペランド**という）を並べれば命令は完成します．その結果，１つの動作をする命令が2バイトあるいは3バイトで構成されるので，それらを称して〝2バイト命令〞，〝3バイト命令〞などということもあります(6809には1〜5バイト命令が存在する)．

Figure-7.1.3の命令表でLDYとLDSの行を見ると，この2行だけ4桁になっていますが，これはLDYとLDSのOPコードが2バイトで構成されるからで，上の例に示したLDYのように〝１０　８Ｅ〞とそのまま並べればよいのです．

もう１つ，アドレッシングモードがインデックスの場合についてですが，Figure-7.1.3の〝LDU　，X〞のアセンブル結果が〝ＥＥ〞ではなく〝ＥＥ８４〞になっていることに気が付かれたでしょうか．これはインデックスモードが24種もあるので，それらを区別するためOPコードのほかにもう1バイト必要になってくるためです．この1バイトのことを**ポストバイト**[*1]といい，インデックスモードのときは必ずこれが必要になります．

当面はインデックスモードといえば〝，Ｘ〞のみを考えていてもらって結構です．したがって，ポストバイトはいつも＄８４としてください．

＊1 ポストバイトについては14章で詳しく解説する

## “ST”

ST(STore)命令は，LD命令のちょうど反対の動作をします．すなわちレジスタA，B，D，X，Y，U，Sの内容を，指定したアドレスに書き込む命令です．ST命令とLD命令の違う点は，データの移動方向が逆になることと，メモリに書くという意味からイミディエイトモードが存在しないことの2点だけです．イミディエイトモードはアドレスを指定するものではなく，データをそのまま指定するものですから，ST命令で使用できないのは当り前のことです．

Figure-7.1.4は命令表1のST命令の部分です．アドレッシングモードに注意して，この図を見ながら次のハンドアセンブルが正しいことを確認してください．

```
97 ●●          ←―――――― STA   <$●●
ED 84          ←―――――― STD   ,X
BF ●● ●●       ←―――― STX   >$●●●●
10 DF ●● ●●    ← STS   <$●●
```

| ニーモニック | | # | < | , | > |
|---|---|---|---|---|---|
| ST | STA | -- | 97 | A7 | B7 |
| | STB | -- | D7 | E7 | F7 |
| | STD | -- | DD | ED | FD |
| | STX | -- | 9F | AF | BF |
| | STY | -- | 109F | 10AF | 10BF |
| | STU | -- | DF | EF | FF |
| | STS | -- | 10DF | 10EF | 10FF |

Figure-7.1.4　ST命令の命令表(命令表1より)

ST命令もLD命令と同じ点に注意する必要があります．〝STX　>＄6000〞ならば，＄6000番地にXレジスタの上位8ビットが，＄6001番地に下位8ビットがそれぞれストアされます．また，インデックスモードのポストバイトは命令によらず一定なので，〝STD　,X〞は〝ED〞ではなく，〝ED　84〞になります．

# 7・2 命令の動作確認

ここでは，前節で解説したLD, ST命令を使って実際に簡単なプログラムを書き，それをハンドアセンブルして実行してみます．ハンドアセンブルの際には巻末の命令表を参照してください.実行の際は，くどいようですが1章のGコマンドで解説した初期設定を忘れずに行ってください.

### 実習1　1バイトのデータをメモリに書き込む

Aレジスタに定数(1バイトのデータ)をロードして，それを任意のアドレスのメモリにストアしてみましょう．定数を＄2A，転送先メモリのアドレスを＄6038とします.

まず，Aレジスタに定数を入れるのですから，アドレッシングモードはイミディエイト(#)を使います．そして定数の値は＄2Aなので，

LDA　　#＄2A…………イミディエイトモードを使い,1バイトデータをAレジスタにロードする

となります.次にAレジスタのデータをそのままメモリにストアしますが，この場合は＄6038番地という決まったアドレスにストアするので，アドレッシングモードにはエクステンド(>)を使います．つまり，

STA　　>＄6038……エクステンドモードを使い,Aレジスタの内容をメモリにストアする

ということです．そして最後にはプログラムの実行を終えて，モニタに戻らなくてはなりませんから，終わりという意味の命令(SWI)が必要です．SWIという命令はまだ説明していませんが，ここでは命令の意味を理解する必要

はありません．これまでも例としていくつかプログラムを掲載しましたが，どれも終わりにはこの命令が付けてあります．

それでは実習1のプログラムをまとめてみましょう．

```
LDA   #$2A…………Aレジスタに$2Aをロードする
STA   >$6038……Aレジスタの内容を$6038番地にストア
                する
SWI ……………………プログラムを終了し，モニタに戻る
```

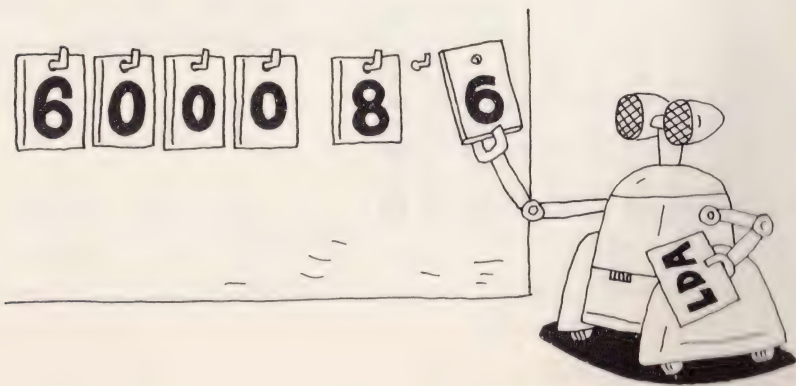
それでは巻末の命令表1を参照しながら，このプログラムをハンドアセンブルしてみましょう．SWIはこの表には載っていませんが，SWIをアセンブルすると＄３Ｆになります．なおプログラムは＄６０００番地から始まるものとします．

```
アドレス   オブジェクト・プログラム   ソース・プログラム
6000       86 2A                      LDA   #$2A
6002       B7 60 38                   STA   >$6038
6005       3F                         SWI
```

一番左の列は各命令の置かれるアドレスを示しており，その次が肝心のアセンブル結果です．

さて，これでアセンブルが完了しましたから，後はこれをコンピュータに入力して実行させるだけです．アセンブルされたマシン語は，行ごとにそのまま続けて入力します．Figure-7.2.1 は実行方法とその結果を確認したものです．

Gコマンドでプログラムを実行させてから＄6038番地を見ると，確かに＄2Aがストアされていることがわかります．

Figure-7.2.1　実習1（1バイトのデータをメモリに書き込む）の実行

###  実習2 メモリ間の1バイトデータの転送

次はインデックスモードを理解するために，任意のアドレスのメモリ内容をXレジスタで指されたアドレスへ転送してみましょう．なお転送するデータは2バイトとします．

まず，転送先となるアドレスをXレジスタで指すためには，そのアドレスをあらかじめXレジスタに入れておかなければなりません．転送先のアドレスを＄６１３０番地とすると，Xレジスタに＄６１３０という値をロードします．つまり＄６１３０は定数ですから，イミディエイトモード(#)を使ってロードすることになります．これは，

**ＬＤＸ　　＃＄６１３０……イミディエイトモードを使い，2バイトデータをXレジスタにロードする**

と書くことができ，これでXレジスタへのアドレスセットは完了します．

次に転送元となるデータが必要ですが，6809には〝あるアドレスの内容をXレジスタの指すアドレスへ転送する〞という命令はないので，この動作を2つの命令に分けて実現します．すなわち，メモリ上のデータをレジスタを経由させて他のメモリへ転送するのです．また，ここでは2バイトのデータを転送するので，ひとまず16ビットの長さを持つDレジスタに転送元のデータをロードします．転送元のデータのあるアドレスを＄６１０２番地とすると，その内容をDレジスタにロードするにはエクステンドモードを使って，

**ＬＤＤ　　＞＄６１０２……エクステンドモードを使い，指定したアドレスの内容をDレジスタにロードする**

と書きます．これで＄６１０２，＄６１０３番地の内容がDレジスタに転送されるわけです．Dレジスタは16ビットのレジスタですから，このように書けば＄６１０３番地も同時に指定したことになります．具体的には＄６１０２番地の内容がAレジスタに，＄６１０３番地の内容がBレジスタに転送されるのですが，LDDという命令はこの2つを一括して扱うことができます．

さて，これでXレジスタの指すアドレスへストアするデータがDレジスタにはいったので，後はST命令を書くだけです．つまり，

STD　,X………………インデックスモードを使い，Xレジスタで指定したアドレスとその次のアドレスへDレジスタの内容をストアする

でよいのです．ここでXレジスタの内容は，さきほどセットした＄６１３０という値がはいっているので，〝STD　,X〞とはAレジスタの内容を＄６１３０番地に，Bレジスタの内容を＄６１３１番地にストアすることを意味しています．インデックスモードのときも，16ビットのデータを扱う命令では暗黙のうちに2つのアドレスを指定したことになるのです．

これで目的の動作をすることができますので，これらをまとめてみましょう．またプログラムの最後はSWI命令で終わらなくてはならないことは，いつも同じです．

LDX　＃＄６１３０……Xレジスタに＄６１３０をロードする
LDD　＞＄６１０２……Dレジスタに＄６１０２，＄６１０３番地の内容をロードする
STD　,X………………Dレジスタの内容を＄６１３０，＄６１３１番地にストアする
SWI…………………………プログラムを終了して，モニタに戻る

それでは命令表1を見ながら上のプログラムをハンドアセンブルしてください．こんどは＄６１００番地をプログラムのスタート・アドレスとします．

| アドレス | オブジェクト・プログラム | ソース・プログラム |
|---|---|---|
| 6100 | 8E 61 30 | LDX #$6130 |
| 6130 | FC B0 00 | LDD >$B000 |
| 6106 | ED 84 | STD ,X |
| 6108 | 3F | SWI |

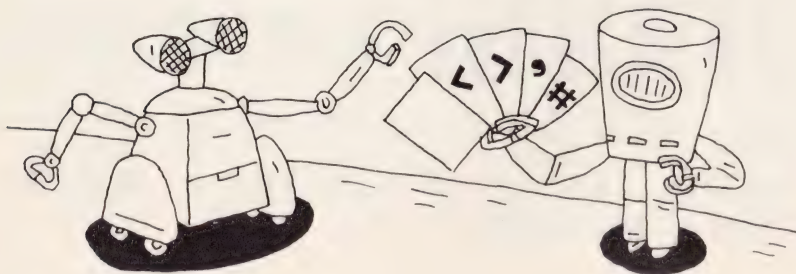
ハンドアセンブルできたら，オブジェクト・プログラムを入力して，実行結果を確認してみましょう．

Figure-7.2.2 実習2(メモリ間の1バイトデータの転送)の実行

```
*M6100  ……$6100番地からプログラムを入力する
6100 00-8E
6101 00-61   "LDX   #$6130"
6102 00-30
6103 00-FC
6104 00-61   "LDD   >$6102"
6105 00-02
6106 00-ED   "STD   ,X"
6107 00-84
6108 00-3F  …"SWI"
6109 00-.

*D6100  ………入力したプログラムを確認する

6100 8E 61 30 FC 61 02 ED 84 ——入力したプログラム
6108 3F 00 00 00 00 00 00 00
6110 00 00 00 00 00 00 00 00
6118 00 00 00 00 00 00 00 00
6120 00 00 00 00 00 00 00 00
6128 00 00 00 00 00 00 00 00
6130 00 00 00 00 00 00 00 00 プログラムの実行前には$6130，
6138 00 00 00 00 00 00 00 00 $6131番地の内容は$00になっている

*G6100 ………プログラムを実行する

*D6100 ………実行結果を確認する

6100 8E 61 30 FC 61 02 ED 84
6108 3F 00 00 00 00 00 00 00
6110 00 00 00 00 00 00 00 00
6118 00 00 00 00 00 00 00 00
6120 00 00 00 00 00 00 00 00
6128 00 00 00 00 00 00 00 00 プログラムの実行によって書き込まれたデータ
6130 30 FC 00 00 00 00 00 00 $6102，$6103番地の内容と一致している
6138 00 00 00 00 00 00 00 00
*
```

次に“ＬＤＸ　＃＄６１３０”のところを値だけ変えてみて，転送先が変わることを合せて確認してみましょう．

Figure-7.2.3　実習2(メモリ間の1バイトデータの転送)の実行

```
*M6101 ↵
6101 61-61 ↵  ┐
6102 30-20 ↵  ┘ LDXのイミディエイトデータを変更する
6103 FC-. ↵

*D6100 ↵      ………変更したプログラムを確認する

6100 8E 61 20 FC 61 02 ED 84  ── 変更したデータ
6108 3F 00 00 00 00 00 00 00
6110 00 00 00 00 00 00 00 00
6118 00 00 00 00 00 00 00 00
6120 00 00 00 00 00 00 00 00  ── プログラムの実行前には$6120,
6128 00 00 00 00 00 00 00 00     $6121番地の内容は$00になっている
6130 30 FC 00 00 00 00 00 00
6138 00 00 00 00 00 00 00 00
*G6100 ↵      ………プログラムを実行する

*D6100 ↵      ………実行結果を確認する

6100 8E 61 20 FC 61 02 ED 84
6108 3F 00 00 00 00 00 00 00
6110 00 00 00 00 00 00 00 00
6118 00 00 00 00 00 00 00 00
6120 20 FC 00 00 00 00 00 00  ── プログラムの実行によって書き込まれたデータ
6128 00 00 00 00 00 00 00 00     $6102,$6103番地の内容と
6130 30 FC 00 00 00 00 00 00     一致している
6138 00 00 00 00 00 00 00 00
*
```

# 8

## 2項演算命令
## (算術演算, 論理演算, 比較命令)

●LD，ST 命令でレジスタとメモリ間で転送ができるようになれば，後はそれらのデータを活用することです．

BASIC を使ってある計算をするプログラムを作るには，数式をそのまま書けばよいのですが，マシン語ではもっと複雑な手順が必要になります．マシン語では演算１つ１つに別々の命令が用意されており，命令にない演算は複数の命令を組み合せて作らなくてはなりません．

本章では，２つのデータを使う演算（２項演算）を行う命令について解説します．6809では，このような演算に必要なデータを，１つはレジスタに，もう１つはメモリに置いています．前章の知識を応用してマシン語による演算を実習してください．

# 8・1 算術演算 ADD, SUBほか

アキュムレータは，様々な演算ができるという点でほかのレジスタと異なっています．インデックス・レジスタなどでも簡単な演算はできますが，アキュムレータほど多くの命令はありません．ここではアキュムレータを使った加減算の方法について解説します．

本章で扱う演算は2項演算といって，

**<データ1>　演算子　<データ2>　⇨　<結果>**
**例：1+1=2**

のような形をしたものばかりです．6809ではこのような演算を，

**<アキュムレータ>　演算子　<メモリ>　⇨　<アキュムレータ>**

のように処理を行うため，演算の結果は必ずアキュムレータに残ります．

## “ ADD, SUB ”

加算(ADD(アド)：ADDition(アディション))，減算(SUB(サブ)：SUBtract(サブトラクト))はアキュムレータ(A, B, Dレジスタ)で実行できますが，A，Bレジスタでは8ビット，Dレジスタでは16ビットのデータを扱います．次に示したのは，ADD，SUB命令の主なアドレッシングの例ですが，メモリの指定には転送命令とまったく同じアドレッシングモードが使えます(巻末の命令表1を参照のこと)．

ADDA　#$●●…………Aレジスタに$●●という値そのものを足す
ADDB　>$●●●●……Bレジスタに$●●●●番地の内容を足す

ADDD ,X………………Dレジスタに X レジスタの指すアドレスの内容を足す
SUBA >$●●●●……Aレジスタから$●●●●番地の内容を引く
SUBB ,X………………Bレジスタから X レジスタの指すアドレスの内容を引く
SUBD #$●●●●……Dレジスタから$●●●●という値そのものを引く

このように書いた場合，結果はどれもそのアキュムレータに残ります．つまり，アキュムレータの内容のみが変更されるのです．

## “ADC, SBC”

1バイトや2バイトの加減算なら ADD, SUB 命令だけでできますが，3バイト以上の演算をする場合，加算や減算は，繰り上がりや繰り下がりを考慮しなければならないため，単純におのおのを加減算しても正しい答えは得られません．

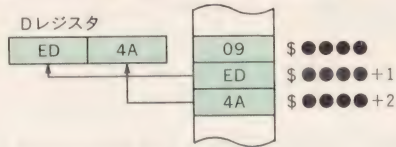
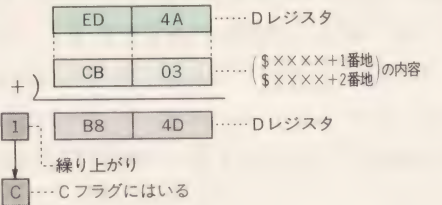
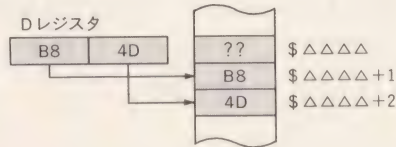
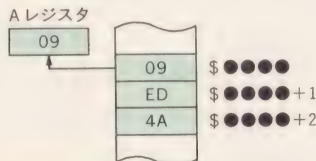
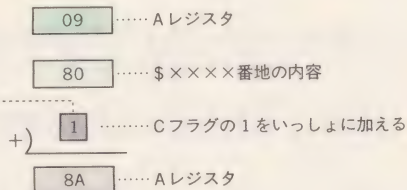
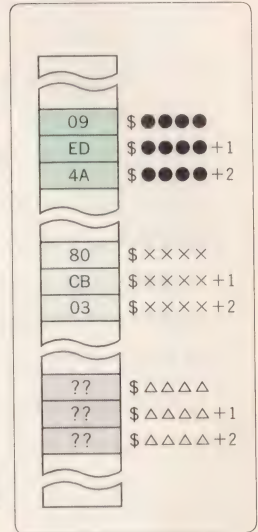
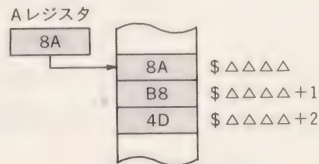
これは，みなさんが筆算で加算を行うときのことを考えれば，すぐに理解できると思います．一番下の桁から順に1桁ずつ加えていって，ある桁で繰り上がりがあったら，次の桁のときにそれも一緒に加えるように，マシン語で多バイト長の加算をするときも，これとまったく同様にして計算するのです．

こういった繰り上がり／繰り下がりが起こる加減算を行うための命令が，ADC（ADdition with Carry：キャリー*1付き加算），SBC（SuBtract with Carry：キャリー付き減算）です．

Figure-8.1.1 に3バイトの加算例を示しましたので，これにそってキャリー付き加減算の説明を行いましょう．

まず下から2バイトを ADDD 命令で加えます．次に3バイト目を加えますが，このとき ADD 命令ではなく ADC 命令で行います．こうすると，初めに2バイト加えたときの繰り上がりが同時に加算されるのです．

*1 演算の結果，繰り上がり／繰り下がりが起こったことを示すフラグ

メモリ上の$●●●●番地からの3バイトと$××××番地からの3バイトの加算を行い，結果を$△△△△からの3バイトにストアする
(1) LDD >$●●●●+1
Dレジスタに$●●●●+1, $●●●●+2 番地の内容をロードする
Dレジスタ
ED
4A
09
ED
4A
$●●●●
$●●●●+1
$●●●●+2
(2) ADDD >$××××+1
Dレジスタの内容と$××××+1, $××××+2番地の内容を足して，Dレジスタに残す
計算の結果発生した繰り上がりはCCレジスタのCフラグにはいる
ED
4A
……Dレジスタ
CB
03
……($××××+1番地 $××××+2番地)の内容
+)
1
B8
4D
……Dレジスタ
繰り上がり
C
……Cフラグにはいる
(3) STD >$△△△△+1
Dレジスタの内容を$△△△△+1, $△△△△+2番地にストアする
Dレジスタ
B8
4D
??
B8
4D
$△△△△
$△△△△+1
$△△△△+2
(4) LDA >$●●●●
Aレジスタに$●●●●番地の内容をロードする
Aレジスタ
09
09
ED
4A
$●●●●
$●●●●+1
$●●●●+2
(5) ADCA >$××××
Aレジスタの内容と$×××× 番地の内容とC フラグの1をいっしょに足してAレジスタに残す
09
……Aレジスタ
80
……$××××番地の内容
1
……Cフラグの1をいっしょに加える
+)
8A
……Aレジスタ
(6) STA >$△△△△
Aレジスタの内容を$△△△△番地にストアする
Aレジスタ
8A
8A
B8
4D
$△△△△
$△△△△+1
$△△△△+2
09
ED
4A
$●●●●
$●●●●+1
$●●●●+2
80
CB
03
$××××
$××××+1
$××××+2
??
??
??
$△△△△
$△△△△+1
$△△△△+2

しかし,CPU は繰り上がりがあったことをどうやって覚えているのでしょうか.

1バイトどうし,または2バイトどうしで加算をしたとき,繰り上がる数は0か1です.つまり1ビットのフラグを別に持っていれば,それで解決できてしまいます.6809はこのことを5.1節で紹介したCCレジスタ内のCフラグで覚えています.ADD命令やADC命令を実行したとき,繰り上がりがあればCフラグは1に,なければ0に自動的に変更されます.ですからADC命令を次々と繰り返していけば,何バイトの加算でも可能になるのです.

減算のときも同様に考えることができます.SUB命令やSBC命令のときに繰り下がりがあればCフラグは1に,なければ0になるので,SBC命令を繰り返せばよいのです.

ADC命令,SBC命令はA,Bレジスタに対しては命令が用意されていますが,Dレジスタではできません.それ以外はADD命令,SUB命令と同様ですべてのアドレッシングモードが使えます.

### 実習3　8ビットデータの加減算

メモリに記憶されている1バイトデータの加減算の実習です.最初にモニタのMコマンドで$6220番地と$6221番地に適当なデータを書き込んだ後,2つのアドレスの内容の和と差を求めて,それぞれ$6230,$6231番地に結果をストアします.

```
LDA   >$6220……$6220番地の内容をAレジスタにロード
               する
LDB   >$6220……$6220番地の内容をBレジスタにロード
               する
ADDA  >$6221……Aレジスタの内容と$6221番地の内容を
               足して,結果をAレジスタに残す
SUBB  >$6221……Bレジスタの内容から$6221番地の内容
               を引いて,結果をBレジスタに残す
```

**STD　　>$6230……A，Bレジスタの内容(Dレジスタの内容)を一度に$6230番地からの2バイトにストアする**

**SWI……………………………プログラムを終了してモニタに戻る**

まずAレジスタとBレジスタに$6220番地の内容をロードし，次にAレジスタには$6221番地の内容を加え，Bレジスタからは$6221番地の内容を引きます．それぞれの演算結果はAレジスタ，Bレジスタに残っているので，最後にDレジスタを$6230番地にストアすれば，Aレジスタ，Bレジスタの両方が同時にストアされ，SWI命令で終了します．

プログラムは$6200番地から始まるものとして，ハンドアセンブルした結果を次に示します．

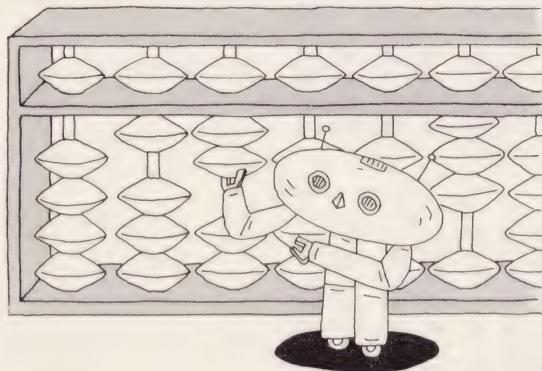
| アドレス | オブジェクト・プログラム | ソース・プログラム |
|---|---|---|
| 6200 | B6　62　20 | LDA　>$6220 |
| 6203 | F6　62　20 | LDB　>$6220 |
| 6206 | BB　62　21 | ADDA　>$6221 |
| 6209 | F0　62　21 | SUBB　>$6221 |
| 620C | FD　62　30 | STD　>$6230 |
| 620F | 3F | SWI |

生成された16バイトのオブジェクト・プログラムをモニタで書き込み，実行した結果を確認してみましょう．

Figure-8.1.2を参考にして＄６２２０，＄６２２１番地に好きな値を入れていろいろと試してください．

Figure-8.1.2　実習3（8ビットデータの加減算）の実行

```
*M6200 ………$6200番地からプログラムを入力する
6200 00-B6
6201 00-62
6202 00-20
6203 00-F6
6204 00-62
6205 00-20
6206 00-BB
6207 00-62
6208 00-21
6209 00-F0
620A 00-62
620B 00-21
620C 00-FD
620D 00-62
620E 00-30
620F 00-3F
6210 00-.

*M6220 ………$6220，$6221番地にデータを入力する
6220 00-C3 ┐加減算のためのデータ
6221 00-21 ┘
6222 00-.

*D6200 ………入力したプログラムとデータを確認する

6200 B6 62 20 F6 62 20 BB 62 ──入力したプログラム
6208 21 F0 62 21 FD 62 30 3F
6210 00 00 00 00 00 00 00 00
6218 00 00 00 00 00 00 00 00
6220 C3 21 00 00 00 00 00 00 ──入力したデータ
6228 00 00 00 00 00 00 00 00
6230 00 00 00 00 00 00 00 00
6238 00 00 00 00 00 00 00 00
*G6200 ………プログラムを実行する

*D6200 ………プログラムの実行結果を確認する

6200 B6 62 20 F6 62 20 BB 62
6208 21 F0 62 21 FD 62 30 3F
6210 00 00 00 00 00 00 00 00
6218 00 00 00 00 00 00 00 00
6220 C3 21 00 00 00 00 00 00
6228 00 00 00 00 00 00 00 00  $E4($6230)=$C3+$21………和
6230 E4 A2 00 00 00 00 00 00  $A2($6231)=$C3-$21………差
6238 00 00 00 00 00 00 00 00
*
```

### 実習4　3バイト以上の加算

繰り上がりの起こる3バイト以上の加算の実習を行います．

＄６３２０～＄６３２２番地と＄６３２８～＄６３２A番地の3バイトどうしを加えて，＄６３３０～＄６３３２番地にその結果をストアするプログラムを書きます．

```
LDD   >$6321……$6321番地のメモリの内容をDレジスタ
                にロードする
ADDD  >$6329……$6329番地のメモリの内容とDレジスタ
                の内容を足し，結果をDレジスタに残す
STD   >$6331……Dレジスタの内容を$6331番地にストアする
LDA   >$6320……$6320番地の内容をAレジスタにロードする
ADCA  >$6328……$6328番地の内容とAレジスタの内容と
                Cフラグの状態(繰り上がり)を一緒に加え，結
                果をAレジスタに残す
STA   >$6330……Aレジスタの内容を$6330番地にストアする
SWI ……………………プログラムを終了し，モニタに戻る
```

初めに下から2バイトをADDDで足し，それを＄６３３１番地へストアしてから一番上の1バイトをADCAによって，繰り上がり(Cフラグ)と一緒に加えます．結果を＄６３３０番地にストアすれば，後はSWI命令で終わりです．

ハンドアセンブルの結果を次に示します．

| アドレス | オブジェクト・プログラム | ソース・プログラム |
|---|---|---|
| 6300 | FC 63 21 | LDD >$6321 |
| 6303 | F3 63 29 | ADDD >$6329 |
| 6306 | FD 63 31 | STD >$6331 |
| 6309 | B6 63 20 | LDA >$6320 |
| 630C | B9 63 28 | ADCA >$6328 |
| 630F | B7 63 30 | STA >$6330 |
| 6312 | 3F | SWI |

ハンドアセンブルしたオブジェクト・プログラムをメモリに書き込んだら，＄６３２０～＄６３２２番地と＄６３２８～＄６３２Ａ番地の各３バイトのメモリには，繰り上がりが起こるようなデータをモニタで書き込んで実行してください．

Figure-8.1.3　実習4（3バイト以上の加算）の実行

```
*M6300↲ ………$6300番地からプログラムを入力する
6300 00-FC↲
6301 00-63↲
6302 00-21↲
  :
6311 00-30↲
6312 00-3F↲
6313 00-.↲

*M6320↲ ………データ$415343を入力する
6320 00-41↲
6321 00-53↲
6322 00-43↲
6323 00-.↲

*M6328↲ ………データ$53D849を入力する
6328 00-53↲
6329 00-D8↲
632A 00-49↲
632B 00-.↲

*D6300↲ ………入力したプログラムとデータを確認する

6300 FC 63 21 F3 63 29 FD 63 ──入力したプログラム
6308 31 B6 63 20 B9 63 28 B7
6310 63 30 3F 00 00 00 00 00
6318 00 00 00 00 00 00 00 00
6320 41 53 43 00 00 00 00 00 ──入力したデータ
6328 53 D8 49 00 00 00 00 00
6330 00 00 00 00 00 00 00 00 ──プログラムの実行前は$00になっている
6338 00 00 00 00 00 00 00 00

*G6300↲ ………$6300番地からプログラムを実行する

*D6300↲ ………Dコマンドで実行結果を確認する

6300 FC 63 21 F3 63 29 FD 63
6308 31 B6 63 20 B9 63 28 B7
6310 63 30 3F 00 00 00 00 00
6318 00 00 00 00 00 00 00 00
6320 41 53 43 00 00 00 00 00
6328 53 D8 49 00 00 00 00 00   プログラムの実行結果
6330 95 2B 8C 00 00 00 00 00    $415343+$53D849
6338 00 00 00 00 00 00 00 00    =$952B8C
*
```

# 8.2 論理演算 AND, OR, EOR

論理演算という言葉を聞き慣れない方もいると思いますが，BASICにもこれと同じような演算があります．ここで扱う論理演算は，AND（論理積），OR（論理和），EOR（Exclusive OR（イクスクルーシブ オア）：論理差，あるいは排他的論理和）と呼ばれるもので，Figure-8.2.1のように定義されている2項演算です．

| 論理積 (AND) | 論理和 (OR) | 論理差 (EOR) |
|---|---|---|
| 0 AND 0 = 0 | 0 OR 0 = 0 | 0 EOR 0 = 0 |
| 0 AND 1 = 0 | 0 OR 1 = 1 | 0 EOR 1 = 1 |
| 1 AND 0 = 0 | 1 OR 0 = 1 | 1 EOR 0 = 1 |
| 1 AND 1 = 1 | 1 OR 1 = 1 | 1 EOR 1 = 0 |

Figure-8.2.1　2項演算

この演算は2進数1桁，すなわち〝0〞か〝1〞しかない世界での演算です．ANDはどちらも1のときのみ演算結果は1となり，ORはどちらか一方が1ならば1となるような演算です．EORは，両方の値が異なるとき1になり，等しいとき0になります．Figure-8.2.2はAND命令の実行例ですが，この図からもわかるようにこれらの演算は1バイト単位で行われます．

Aレジスタの内容が$39のとき，イミディエイトデータ$0FとANDをとる

ANDA #$0F

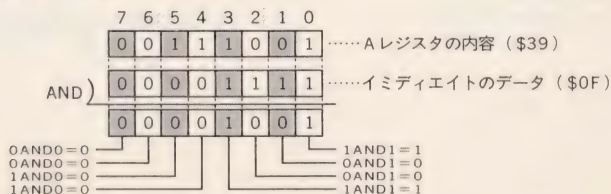



Figure-8.2.2　論理演算(AND)の命令動作

そのため8ビットが表す数値の意味はまったくなく，各ビットごとにしか意味を持ちません．

論理演算は，コンピュータのハードウェアを構成するためには非常に重要なものです．しかし，ソフトウェアとなるとこれらの命令の使用頻度はあまり高くなく，計算という意味あいよりも，ビットごとのセット／リセット／反転の目的で使用されることが多いのです．例えば，現在のアキュムレータの内容のビット0からビット3までを残し，上位4ビットをすべて0にする場合，$0F(2進数では00001111)とANDをとれば，上位4ビットは0になり，下位4ビットは変わりません．同様に，1にしたいビットがあればそのビットだけ1でほかは0であるような値とORをとり，そのビットを反転(0ならば1，1ならば0)するのであれば，EORをとります．ちょっとわかりづらいかもしれませんが，Figure-8.2.1を参考にしてFigure-8.2.3に示した各論理演算の結果を確認してください．

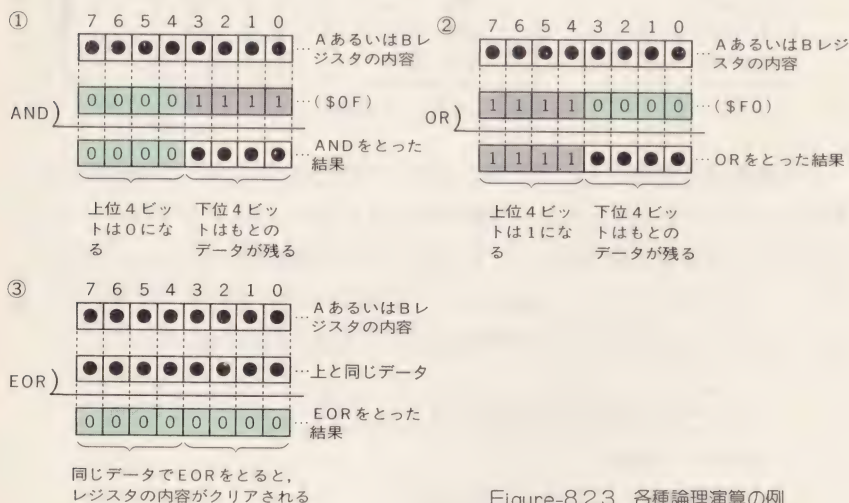



Figure-8.2.3　各種論理演算の例

6809におけるこれらの演算はA，Bレジスタに対して可能ですが，Dレジスタに対する命令は用意されていません．また，論理演算には繰り上がりなどないので，ADCのような命令も必要ありません．アドレッシングモードはいままでと同様，4つのモードすべてが使えます．以下論理演算命令の例をいくつかあげましょう．

ANDA　#＄0F…………Aレジスタと＄0FのANDをとってAレジスタに結果を残す
ORB　>＄E804……Bレジスタと＄E804番地のメモリの内容とのORをとりBレジスタに結果を残す
EORA　,X………………Xレジスタの指すアドレスの内容とAレジスタとのEORをとりAレジスタに結果を残す

実習5　8ビットデータの論理演算

メモリの1バイトデータを使ってAND命令の実習を行います．

＄6420番地と＄6421番地の内容の論理積(AND)を＄6422番地にストアするプログラムを＄6400番地から書いてみましょう．

LDA　>＄6420……＄6420番地の内容をAレジスタにロードする
ANDA　>＄6421……＄6421番地の内容とAレジスタの内容とのANDをとり，結果をAレジスタに残す
STA　>＄6422……Aレジスタの内容を＄6422番地にストアする
SWI…………………………プログラムを終了して，モニタに戻る

ハンドアセンブルの結果を次に示します．

| アドレス | オブジェクト・プログラム | ソース・プログラム |
|---|---|---|
| 6400 | B6 64 20 | LDA >$6420 |
| 6403 | B4 64 21 | ANDA >$6421 |
| 6406 | B7 64 22 | STA >$6422 |
| 6409 | 3F | SWI |

オブジェクト・プログラムを入力して，実行結果を確認してみましょう．

Figure-8.2.4 実習5（8ビットデータの論理演算）の実行

```
*M6400↵ ……$6400番地からプログラムを入力する
6400 00-B6↵
6401 00-64↵
6402 00-20↵
  :
6408 00-22↵
6409 00-3F↵
640A 00-.↵

*M6420↵ ……ANDをとる2つのデータを入力する
6420 00-C3↵
6421 00-8F↵
6422 00-.↵

*D6400↵ ……入力したプログラム，データを確認する

6400 B6 64 20 B4 64 21 B7 64 ——入力したプログラム
6408 22 3F 00 00 00 00 00 00
6410 00 00 00 00 00 00 00 00
6418 00 00 00 00 00 00 00 00 ——入力したデータ
6420 C3 8F 00 00 00 00 00 00 ——プログラムの実行前は$00になっている
6428 00 00 00 00 00 00 00 00
6430 00 00 00 00 00 00 00 00
6438 00 00 00 00 00 00 00 00
*G6400↵ ……プログラムを実行する

*D6400↵ ……実行結果を確認する

6400 B6 64 20 B4 64 21 B7 64
6408 22 3F 00 00 00 00 00 00
6410 00 00 00 00 00 00 00 00
6418 00 00 00 00 00 00 00 00
6420 C3 8F 83 00 00 00 00 00 ——$6420番地から入力した
6428 00 00 00 00 00 00 00 00    2バイトのデータのANDをとった結果
6430 00 00 00 00 00 00 00 00          $C3……11000011
6438 00 00 00 00 00 00 00 00    AND) $8F……10001111
*                                      $83……10000011
```

# 8.3 比較 CMP, BIT

比較命令の演算自体の意味は算術演算や論理演算と同じですが，演算結果がアキュムレータに残らず，フラグだけがこれまでの演算命令と同じ影響を受けます．つまり，比較命令を使う目的は演算結果にあるのではなく，演算結果によって変化するフラグの状態にあるのです．プログラムのなかでは，いろいろな条件によってその流れを制御していますが，フラグの状態は，その判断材料として利用されるのです．

比較命令の本来の意味を理解するには，条件判断についての解説が不可欠なので，本章では命令の解説のみを行い，プログラムの実習は13章の条件分岐命令のところに譲ることにします．

## “CMP”

CMP(CoMPare(コンペア))命令は，結果がレジスタに残らないことを除けば，SUB命令とまったく同じ動作をします．つまりこの命令を実行すると，繰り下がりなどの情報がフラグにセットされるだけなのです．その際，Cフラグのほかに，まだ説明していないZフラグやNフラグなども同時に変更され，大小比較などに利用されます．CMP命令とSUB命令のフラグの動きが同じであることを，命令表1を見て確認しておいてください．

また，CMP命令の使用できるアドレッシングモードはSUB命令などと同じですが，アキュムレータだけでなくX，Y，U，Sなどのレジスタに対しても使用できます．

CMP命令の使用例を次にいくつか示しておきます．

CMPA ,X……………………Aレジスタの内容からXレジスタの指すアドレスの内容を引き，その演算結果によってフラグに影響を与える

CMPX #$●●●●……Xレジスタの内容から$●●●●を引き，その演算結果によってフラグに影響を与える

CMPU >$●●●●……Uレジスタの内容から$●●●●番地の内容を引き，その演算結果によってフラグに影響を与える

## “BIT”

BIT(BIt Test)命令は，AND命令と同じ演算をして，CMP命令同様結果をレジスタに残さずフラグのみを変更します．この命令は，いわゆる論理演算をするためのものではなく，特定ビットに対する(1か0かの)テストの意味で使われます．つまりアキュムレータの8ビットと特定のビットのみ1であるような値とAND(BIT)をとれば，演算結果はそのビット以外すべて0になるので，他のビットに影響されずにテストができるのです．

この命令の使えるレジスタ，アドレッシングモードはAND命令とまったく同じです．

BITA #$10……Aレジスタの内容と$10とのANDをとり，その演算結果によってフラグに影響を与える

BITB #$0F……Bレジスタの内容と$0FとのANDをとり，その演算結果によってフラグに影響を与える

本章では，6809でできる2項演算で，アドレッシングモードにエクステンド，ダイレクト，イミディエイト，インデックスの4つが使える命令をすべて紹介しました(STは命令の意味からイミディエイトは使えませんが)．7章，8章で解説した命令は11種にすぎませんが，どの命令も4つのアドレッシングモードと組み合せて使えるので，レジスタの種類も含めればすでに149通りの命令を理解したのに等しいのです．

# 9 単項演算命令（増減,クリア,テスト命令）

●本章，次章を通じて，単項演算命令のグループについて解説します．

数学で使う記号に〝−〟（マイナス）がありますが，これには〝ある値から別の値を引く〟，〝ある値の符号を反転する〟という2つの意味があります．前者は2項演算子，後者は単項演算子で，両者は別の意味を持つものです．

単項演算を一般的に定義すると，

演算子〈データ〉=〈結果〉

のような形の演算で，その演算にはデータが1つだけ必要です．普段あまり聞き慣れない言葉なので，意味がつかみにくいかも知れませんが，本章の命令を見ていけば，おのずと理解できると思います．

# 9.1 算術，論理演算 NEG, COM

ここで紹介する6809の単項演算命令はCOM (COMpliment) とNEG (NEGate)の2つで，それぞれ論理演算，算術演算の命令です．

## “COM”

COM命令はAND, ORなどの論理演算命令の仲間で，データのNOTすなわち否定をとります．AND, ORがそうであったように，この命令も8ビットをそれぞれ独立に扱い，Figure-9.1.1に示すように各ビットを反転します．つまり，もともと0であったビットは1に,1であったビットは0になります．

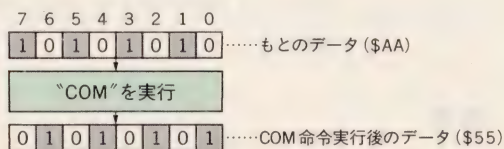



Figure-9.1.1　COM命令の動作

## “NEG”

NEG命令はADD, SUBなど算術演算命令の仲間で，2の補数で考えたときの8ビットデータを，絶対値を変えずに符号反転します．ここで，“2の補数”という言葉が出てきましたが，これはコンピュータで扱う数値データの表現法の1つです．詳しくはのちほど解説しますので，まずNEG命令の動作を見てみましょう．Figure-9.1.2にNEG命令の動作を図解します．

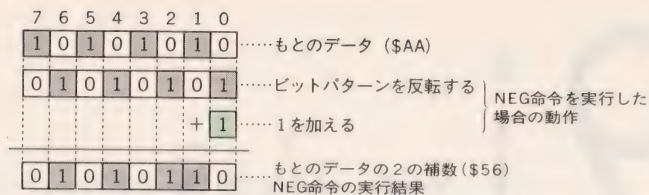



Figure-9.1.2 NEG命令の動作

これらの命令にデータとして与えられるものは，A，Bレジスタ，さらにエクステンド，ダイレクト，インデックスの各アドレッシングモードで表される任意のメモリです．そして，その演算結果はもとの場所(レジスタやメモリ)にはいります．

次にこれらの使用例を示しましょう．

COM　　>$●●●●……$●●●●番地の内容の各ビットを反転する
COMB……………………Bレジスタの内容の各ビットを反転する
NEGA……………………Aレジスタの内容の符号を反転する
NEG　　，X……………Xレジスタの指すアドレスの内容の符号を反転する

## 2の補数

**2の補数**とは，2進数で正，負の数を表す方法の1つです．8ビット，すなわち2進数8桁では$00～$FFの256通りの数が表せますが，その半分の128通りの場合を負の数に割り当てます．具体的には$00～$7Fを正，$80～$FFが負になります．この意味は次のように考えることによって簡単に理解できます．

8ビットのレジスタに，初め$00がはいっているとして，それに1を加えて行くことを考えます．するとそのレジスタは$01，$02，$03，…と増えて行きますが，それが$FD，$FE，$FFまできたとき，次に1を加えたらどうなるでしょうか．本来は$100のように9ビット目に繰り

上がりが出ますが，レジスタは8ビットしかないので繰り上がりは無視され，＄００に戻ってしまいます．

一方，負の数とは0よりも小さい数のことですから，たとえば〝−1″を1加えたときに0になる数ということもできます．つまり−nという数は，nを加えたら0になる数のことです．

さて，このように負の数を考えると，さきほどの＄ＦＦはどのように考えることができるでしょうか．＄ＦＦは1を加えたとき＄００になりますから，0から見ると1つ手前，すなわち1つ小さい数と見ることができます．つまり，＄ＦＦを−1と考えても矛盾は起こりません．同様に＄ＦＥは−2，＄FDは−3と見ることもできます．

このようにして＄８０〜＄ＦＦを−128〜−1，＄００〜＄７Ｆを0〜＋127とみなすことによって，8ビットで正，負の数を表すことができ，このことを2の補数といいます[*1]．つまり2の補数で表されたデータは，ビット7(最上位ビット)が1のとき負，0のとき正という性質を持っています．

2の補数では，＄ＦＦと＄００は連続しており，＄７Ｆと＄８０のところで不連続になります．2の補数を使うメリットは，0の前後で値が連続しているからなのです．Figure-9.1.3は16進数と2の補数の関係を表したものです．

次に，2の補数で表したデータの符号を反転する方法について説明しましょう．Figure-9.1.3で正負の境になっている線に対して，線対称な位置関係にある2つのデータを比較してみます．

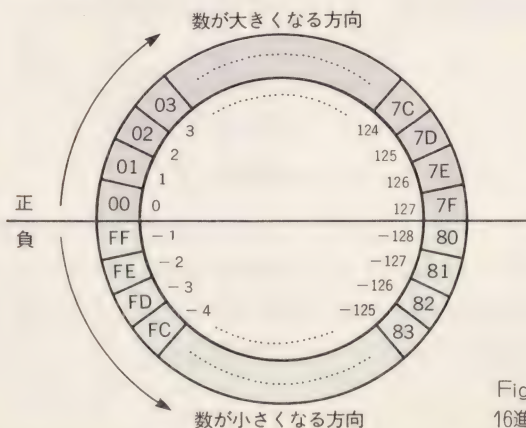



Figure-9.1.3
16進数と2の補数の関係

*1 たがいに2の補数となっている2つのnビット長のデータを加算すると$2^n$になる

例として＄０３と＄ＦＣを取り上げてみますと，この２つを２進数で表したとき，ちょうど各桁(ビット)がお互いに反対の数になっていることに気付きます．両者を足すと＄ＦＦになりますが，このことは＄０３と＄ＦＣに限ったことではなく，すべての線対称な位置にあるデータの組に対して成り立ちます．また＄ＦＦは１加えると０になる数ですから，これらの２つの数を<データ1>，<データ２>で表せば，次のような関係が成立します．

<データ１>＋<データ２>＋1＝0

これを書き直すと，

−<データ１>＝<データ２>＋1

となりますが，これは<データ１>の符号を反転する方法を示しています．つまり，−<データ１>を求めたければ，まず<データ２>を求めて，それに１を加えればよいのです．<データ２>は，<データ１>のすべてのビットを反転すれば簡単に求めることができます．例として＄０３の符号を反転する様子を Figure-9.1.4 に図示しておきます．

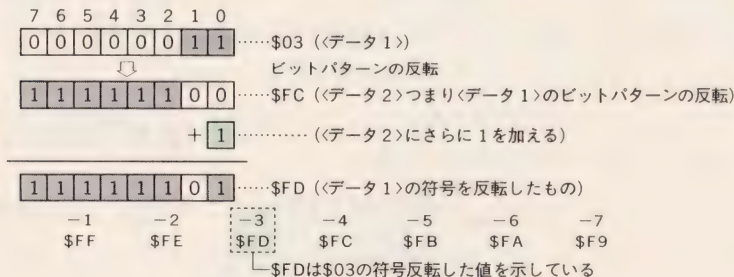



Figure-9.1.4 16進数($03)の符号反転

 実習６　８ビットデータの２の補数をとる

メモリ上の１バイトデータを使って，COM 命令と NEG 命令の働きを実習します．プログラムでは，＄６５２０番地の内容の各ビットを反転し，さらに＄６５２１番地の内容の符号を反転させてみましょう．COM，NEG 命令ともエクステンドモードでアドレスを指定します．

スタート・アドレスを＄６５００番地としてハンドアセンブルしたものを次に示します．

```
6500   73 65 20   COM   >$6520
6503   70 65 21   NEG   >$6521
6506   3F         SWI
```

モニタでオブジェクト・プログラムを入力したら，＄６５２０番地と＄６５２１番地に同じデータをセットして，COM，NEG命令の動作を確認してください．Figure-9.1.5では，＄３Ａをセットしています．

Figure-9.1.5　実習6(8ビットデータの2の補数をとる)の実行

```
*M6500↲………プログラムを入力する
6500 00-73↲
6501 00-65↲
6502 00-20↲
6503 00-70↲
6504 00-65↲
6505 00-21↲
6506 00-3F↲
6507 00-.↲

*M6520↲………データを入力する
6520 00-3A↲
6521 00-3A↲
6522 00-.↲

*D6500↲………入力したプログラムとデータを確認する
                                        入力したプログラム
6500 73 65 20 70 65 21 3F 00
6508 00 00 00 00 00 00 00 00
6510 00 00 00 00 00 00 00 00
6518 00 00 00 00 00 00 00 00
6520 3A 3A 00 00 00 00 00 00  入力したデータ．このデータのCOMとNEGをとる
6528 00 00 00 00 00 00 00 00
6530 00 00 00 00 00 00 00 00
6538 00 00 00 00 00 00 00 00
*G6500↲………プログラムを実行する

*D6500↲………プログラムの実行結果を確認する

6500 73 65 20 70 65 21 3F 00
6508 00 00 00 00 00 00 00 00
6510 00 00 00 00 00 00 00 00
6518 00 00 00 00 00 00 00 00     ($C5は$3Aのビットパターンを反転した結果)
6520 C5 C6 00 00 00 00 00 00  $3A        COM  $C5
6528 00 00 00 00 00 00 00 00  00111010  ──→  11000101
6530 00 00 00 00 00 00 00 00  $3A        NEG  $C6($C5+1)
6538 00 00 00 00 00 00 00 00  00111010  ──→  11000110
*                                  ($3Aと$C6は互いに2の補数)
```

# 9.2 増減命令 INC, DEC

マシン語に限らずコンピュータのプログラムを書いていると，レジスタやメモリの内容に 1 を加えるとか，1 を引くという動作がよく出てきます．これらはもちろん ADD 命令，SUB 命令で実現できますが，頻繁に出てくることなので専用の命令として INC 命令(INCrement : 1 を加える)と DEC 命令(DECrement : 1 を引く)が用意されています．

命令の意味を数学的にいうならば，あくまで 2 項演算なのですが，データの一方が定数であり，命令に与えられるデータが 1 つであることから，あえて単項演算のグループに入れてあります．

巻末の命令表 2 を見ればわかるように，INC／DEC 命令も COM／NEG 命令とまったく同様にデータの指定ができます．つまりメモリに対しても直接インクリメント／デクリメントができるので，LD 命令でアキュムレータにデータをロードしてきて，ADD 命令や SUB 命令を使って計算した後，ST 命令でメモリに戻す必要はないのです．

次に INC，DEC 命令の例をいくつか示すとともに Figure-9.2.1 にその動作を図解しておきます．

```
INCA…………………………A レジスタの内容をインクリメント(+1)する
DEC    >$●●●●……$●●●●番地の内容をデクリメント(−1)する
INC    ,X…………………X レジスタの指すアドレスの内容をインクリメント(+1)する
DECB…………………………B レジスタの内容をデクリメント(−1)する
```

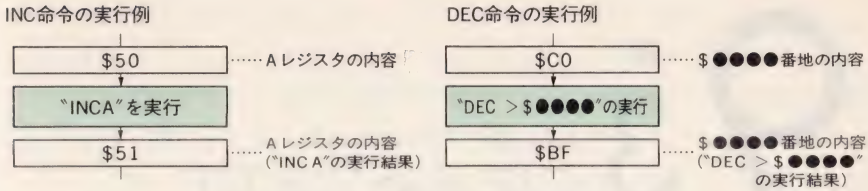



Figure-9.2.1 INC，DEC命令の動作

Figure-9.2.1を見る限りでは，INC，DEC命令はADD，SUB命令で置き換えることができるように思えますが，両者の働きはまったく同じではありません．

例えば，Aレジスタの内容が＄ＦＦであった場合，

ＩＮＣＡ……………………………Aレジスタが＄００になる

と

ＡＤＤＡ　#1 ………………Aレジスタが＄００になる

ではどちらもAレジスタに残る結果は同じですが，命令実行後のフラグに与える影響が異なっています．この場合，ADD命令では繰り上がりがあるのでCフラグがセットされますが，INC命令ではCフラグに影響を与えません．この関係は，DEC命令とSUB命令でも同様です．Aレジスタの内容が＄００のときに，

ＤＥＣＡ……………………………Aレジスタが＄ＦＦになる

と

ＳＵＢＡ　#1 ………………Aレジスタが＄ＦＦになる

のAレジスタに残る結果は同じですが，やはり演算の結果に繰り下がりがあるため，SUB命令ではCフラグがセットされますが，DEC命令ではフラグに影響を与えません．

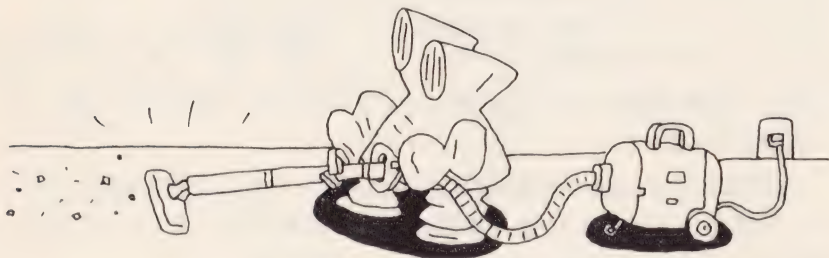
ここで述べたことは，いまはまだピンとこないかもしれませんが，プログラムを組むようになると，ある程度注意する必要があります．なお，INC，DEC命令の実習は，次節の命令といっしょに行います．

# 9・3 クリア，テスト CLR, TST

プログラムでは，データの初期値として0をセットする場合がよくありますが，マシン語も例外ではありません．レジスタに0を入れることは，LD命令のイミディエイトモードでもできますが，6809には専用の命令としてCLR（CLeaR（クリア））命令が用意されています．この命令も，ほかの単項演算命令と同様アキュムレータ，メモリに対して実行できます．

CLR命令の例をいくつか示しておきましょう．

CLRA……………………………Aレジスタに0を入れる
CLR　　＞$●●●●……$●●●●番地のメモリに0を入れる
CLR　　，X………………Xレジスタの指すアドレスのメモリに0を入れる

CLR命令の特徴は，この命令が実行されたときのフラグの変化が常に一定であることです．フラグに関する説明は13章で行いますが，ひとまずCLR命令のフラグの変化を巻末の命令表2で確認してみましょう．この命令を実行した後は，N，Z，Cの3つのフラグは必ず0，1，0になります．

TST(TeST)命令は8章で出てきたCMP命令やBIT命令などの仲間で，指定されたアドレスの内容やデータから0を引いて，単にフラグを変えるためだけの命令です．この命令が取り得るアドレッシングモードもほかの単項演算命令と同じなので，あえてこの章に分類しました．

TSTA……………………………Aレジスタの内容から0を引いて，フラグに影響を与える

TST　　>$●●●●……$●●●●番地の内容から0を引いて，フラグに影響を与える

この命令は，主に条件分岐命令とともに使われるので，実習は13章で行います．ここではTST命令の実行によって，フラグがどのように変化するかを図解しておきます．

| | フラグ N Z C | |
|---|---|---|
| ① Aレジスタの内容が$00の場合<br>TSTAを実行すると<br>Aレジスタ($00)−0=0 | 0 1 · | Zフラグが1なのでAレジスタの内容が$00であることがわかる |
| ② $●●●●番地の内容が$FFの場合<br>TST　>$●●●●を実行すると<br>$●●●●番地($FF)−0=$FF | 1 0 · | Nフラグが1なので$●●●●番地の内容は2の補数で考えた場合には負の数であることがわかる |
| ③ Xレジスタの指すアドレスの内容が$0Cの場合<br>TST　,Xを実行すると<br>$0C−0=$0C | 0 0 · | Nフラグが0なので，Xレジスタの指すアドレスの内容は2の補数で考えた場合には正の数であることがわかる |

Figure-9.3.1　TST命令の動作

Figure-9.3.1を見ると，TST命令に与えられるデータによってフラグがいろいろと変化していることがわかります(Cフラグは変化しない)．この図では，TST命令実行時のフラグの変化を明確にするために，あらかじめレジスタやメモリの内容を明示してありますが，プログラムのなかではこれらの内容はいつどのような値がはいっているかはわかりません．つまり，プログラム実行中に扱うデータがその実行とともに次々と変化していくような場合，TST命令実行後のフラグを参照すれば，そのデータがどのような数値であるかを区別することができるのです．

##  実習７　メモリのクリア／インクリメント／デクリメント

CLR, INC, DEC命令をメモリの１バイトデータに対して実行し，それぞれの命令の働きを実習します．

命令ごとに，次のようなプログラムを書いてみましょう．

① ＄６６３０番地の内容を，クリア(0に)するプログラム
② ＄６６３０番地の内容を，インクリメント(+1)するプログラム
③ ＄６６３０番地の内容を，デクリメント(−1)するプログラム

上の３つをそれぞれ＄６６００，＄６６１０，＄６６２０番地から別のプログラムとして書き，ハンドアセンブルした結果を次に示します．

```
6600   7F 66 30   CLR  >$6630  }
6603   3F         SWI          } ①
          ⋮
6610   7C 66 30   INC  >$6630  }
6613   3F         SWI          } ②
          ⋮
6620   7A 66 30   DEC  >$6630  }
6623   3F         SWI          } ③
```

すべて１命令でできるので，プログラムとは呼べないくらいですが，これらをモニタで入力し，実行結果を確認してみましょう．プログラムの実行前には，あらかじめ＄６６３０番地に任意の１バイトデータをセットしておいてください．

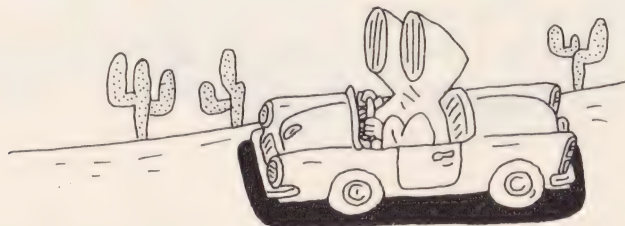
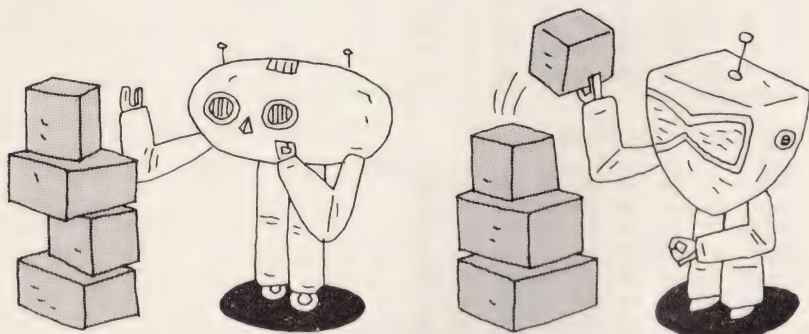



Figure-9.3.3　実習7（メモリのクリア／インクリメント／デクリメント）の実行

```
*M6600↲………＄６６３０番地の内容をクリアするプログラムを入力する
6600 00-7F↲
6601 00-66↲
6602 00-30↲
6603 00-3F↲
6604 00-.↲

*M6610↲………＄６６３０番地の内容をインクリメント（＋１）するプログラムを入力する
6610 00-7C↲
6611 00-66↲
6612 00-30↲
6613 00-3F↲
6614 00-.↲

*M6620↲………＄６６３０番地の内容をデクリメント（－１）するプログラムを入力する
6620 00-7A↲
6621 00-66↲
6622 00-30↲
6623 00-3F↲
6624 00-.↲

*M6630↲………＄６６３０番地にデータ＄１Ａを書き込む
6630 00-1A↲
6631 00-.↲

*D6600↲………入力したプログラム，データを確認する
                                   ―――CLR
6600 7F 66 30 3F 00 00 00 00
6608 00 00 00 00 00 00 00 00
6610 7C 66 30 3F 00 00 00 00       ―――INC
6618 00 00 00 00 00 00 00 00
6620 7A 66 30 3F 00 00 00 00       ―――DEC
6628 00 00 00 00 00 00 00 00
6630 1A 00 00 00 00 00 00 00       ―――データ
6638 00 00 00 00 00 00 00 00
*G6610↲………インクリメントするプログラムを実行する

*D6600↲………結果を確認する

6600 7F 66 30 3F 00 00 00 00
6608 00 00 00 00 00 00 00 00
6610 7C 66 30 3F 00 00 00 00
6618 00 00 00 00 00 00 00 00
6620 7A 66 30 3F 00 00 00 00
6628 00 00 00 00 00 00 00 00
6630 1B 00 00 00 00 00 00 00
6638 00 00 00 00 00 00 00 00       ―――$1A+1=$1B
*G6620↲………デクリメント（－１）するプログラムを実行する
```

```
*D6600↵……結果を確認する

6600 7F 66 30 3F 00 00 00 00
6608 00 00 00 00 00 00 00 00
6610 7C 66 30 3F 00 00 00 00
6618 00 00 00 00 00 00 00 00
6620 7A 66 30 3F 00 00 00 00
6628 00 00 00 00 00 00 00 00
6630 1A 00 00 00 00 00 00 00 ―――$1B−1=$1A
6638 00 00 00 00 00 00 00 00
*

*G6600↵……クリアするプログラムを実行する

*D6600↵……結果を確認する

6600 7F 66 30 3F 00 00 00 00
6608 00 00 00 00 00 00 00 00
6610 7C 66 30 3F 00 00 00 00
6618 00 00 00 00 00 00 00 00
6620 7A 66 30 3F 00 00 00 00
6628 00 00 00 00 00 00 00 00
6630 00 00 00 00 00 00 00 00 ―――クリアされた
6638 00 00 00 00 00 00 00 00
*
```

# 10 シフト／ローテート命令

●マシン語の算術演算命令には，加減算命令と8ビットの乗算命令がありますが，これらの命令だけでは満足な演算が行えません．加減算の繰り返しによって乗除算を行うこともできますが，シフト／ローテート命令を使って2進数のビットパターンを全体に左右にずらし，2倍あるいは1/2のデータを得るというのがマシン語での一般的な方法です．これは，実際に2進数を左右にずらしたものを10進数に変換してみればすぐにわかります．

本章では，シフト／ローテート命令の解説を行いますが，特に乗除算の方法は，2進数の性質をうまく利用していますので，注意して読んでください．

シフト／ローテート命令は，ビットパターンを左右にずらす命令というよりは，こういった乗除算の力不足を補う演算命令として用意されているといった方がよいでしょう．

# 10i ロジカル・シフト LSL, LSR

**シフト**とはアキュムレータやメモリを単なるビット列とみなして，それを左や右に1つずらすことです(Figure-10.1.1)．

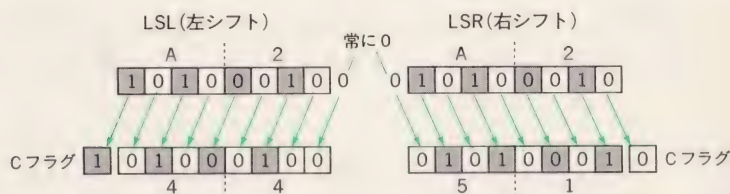



Figure-10.1.1 シフト命令の動作

左シフトの場合，各ビットを左隣へ移します．こうするとビット7があふれますからそれをCフラグへ入れ，またビット0にははいるものがないので0が代入されます．右シフトはこれと逆のことが行われ，ビット0がCフラグにはいり，ビット7には0がはいります．

これらの命令を繰り返しながらCフラグを見ていれば，何番目のビットが1であるか0であるかを知ることができます．またこの8ビットを数値として見れば，左シフトは2倍，右シフトは1/2したことと等しいのです．このことは私たちが10進数を10倍，1/10するときに0を付け加えたり取ったりするのと同じことです．

LSL(Logical Shift Left)命令，LSR(Logical Shift Right)命令も含め，本章で解説する命令は，すべてA, Bレジスタ，エクステンド，ダイレクト，インデックスの各アドレッシングモードが使えます(巻末の命令表2参照)．

LSLA……………………Aレジスタの内容を左へシフトする
LSRB……………………Bレジスタの内容を右へシフトする
LSL　>$●●●●……$●●●●番地の内容を左へシフトする
LSR　,X………………Xレジスタの指すアドレスの内容を右へシフトする

実習8　ロジカル・シフト

メモリの1バイトデータをシフトするLSL命令の実習を行います．次に示したプログラムは，$6730番地にあるデータを2回左シフトした値を$6731番地へ，4回左シフトした値を$6732番地へストアするものです．スタート・アドレスは$6700番地としてハンドアセンブルしています．

```
6700  B6 67 30  LDA  >$6730 ┐ $6730番地の内
6703  48        LSLA        │ 容を2回シフトして
6704  48        LSLA        │ $6731番地にス
6705  B7 67 31  STA  >$6731 ┘ トアする
6708  48        LSLA        ┐ さらにAレジスタの
6709  48        LSLA        │ 内容を2回シフトし
670A  B7 67 32  STA  >$6732 ┘ て$6732番地に
                              ストアする
670D  3F        SWI
```

まずデータをAレジスタに取り込みます．次に左へ2回シフトすると，1つ答えが求まりますから，それをメモリにストアします．さらに左へ2回シフトすれば合せて4回左へシフトしたことになるので，それもメモリにストアすればOKです．

左へ1回シフトすると値は2倍になるので,2回シフトすれば4倍になりますし,4回シフトすれば16倍になります．結果が$FFより大きくならない範囲でなら正しく計算されます．

Figure-10.1.2が，プログラムの入力および実行結果の確認です．

Figure-10.1.2 実習8(ロジカル・シフト)の実行

```
*M6700↲ ……$6700番地からプログラムを入力する
6700 00-B6↲
6701 00-67↲
6702 00-30↲
6703 00-48↲
6704 00-48↲
6705 00-B7↲
6706 00-67↲
6707 00-31↲
6708 00-48↲
6709 00-48↲
670A 00-B7↲
670B 00-67↲
670C 00-32↲
670D 00-3F↲
670E 00-.↲

*M6730↲ ……シフトするデータを入力する
6730 00-91↲
6731 00-.↲

*D6700↲ ……入力したプログラム,データを確認する

6700 B6 67 30 48 48 B7 67 31 ――プログラム
6708 48 48 B7 67 32 3F 00 00
6710 00 00 00 00 00 00 00 00
6718 00 00 00 00 00 00 00 00
6720 00 00 00 00 00 00 00 00
6728 00 00 00 00 00 00 00 00
6730 91 00 00 00 00 00 00 00 ――データ $91をビットパターンで表すと
6738 00 00 00 00 00 00 00 00                 1001|0001
*G6700↲ ……プログラムを実行する

*D6700↲ ……実行結果を確認する

6700 B6 67 30 48 48 B7 67 31
6708 48 48 B7 67 32 3F 00 00
6710 00 00 00 00 00 00 00 00
6718 00 00 00 00 00 00 00 00
6720 00 00 00 00 00 00 00 00
6728 00 00 00 00 00 00 00 00
6730 91 44 10 00 00 00 00 00
6738 00 00 00 00 00 00 00 00
*                                                   4    4
          └$91を4回シフトした値   $91を2回シフト……10|0100|0100|
     └$91を2回シフトした値        $91を4回シフト……1001|0001|0000|
                                                       1    0
```

# 10.2 アリスメティック・シフト ASL, ASR

8ビットのデータを数値として考えると，それをシフトすれば2倍，1/2の値になることは前節で述べましたが，これは厳密にいうと正しくありません．8ビットを2の補数で表現された値とした場合，負の数はビット7が1でなければならないはずですが，それをLSR命令などでシフトするとビット7が0，すなわち正の数になってしまうからです．Figure-10.2.1を見てください．このような不都合は，負の値を1/2にしようとしてLSR命令(右シフト)を行ったときに生じます．

LSR命令で負の数を½にした場合

例えば，Bレジスタの内容が$C4の場合，LSR命令を実行すると次のようになる

$C4(−60)を2の補数として考えた場合には$C4の½は$E2(−30)にならなければならないがLSR命令では$62になってしまう

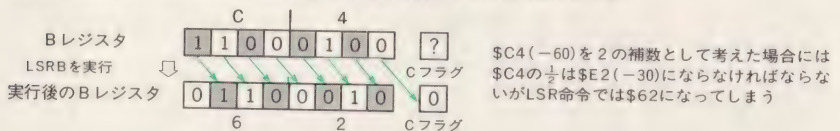



Figure-10.2.1　負の数(2の補数)をLSR命令でシフトした結果

2の補数で表現された値を正しく1/2するためには，LSR命令とは別のシフト命令が必要になります．一方，左シフトをして2倍するときや符号なしの値のときはこんな不都合は起こりません．

ASL(Arithmetic Shift Left)命令，ASR(Arithmetic Shift Right)命令は，これらの矛盾を起こさずに計算するためのシフト命令です．実際には左シフトはLSL命令を使うことができるので，LSL命令とASL命令はアセンブルしてしまえば同じ命令になります．つまり同じ命令をニーモニックでは2通りに書けるということです．巻末の命令表2を見て確認しておいてください．

では，2の補数で表された数値をどのように正しく1/2するのか，ASR命令の動作をFigure-10.2.2で説明しましょう．

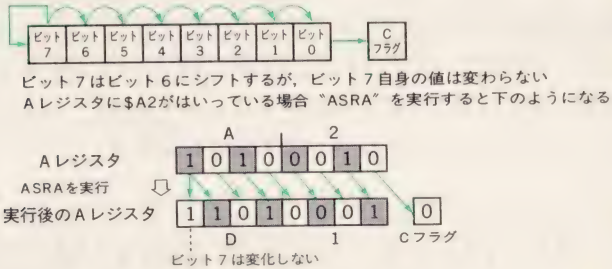
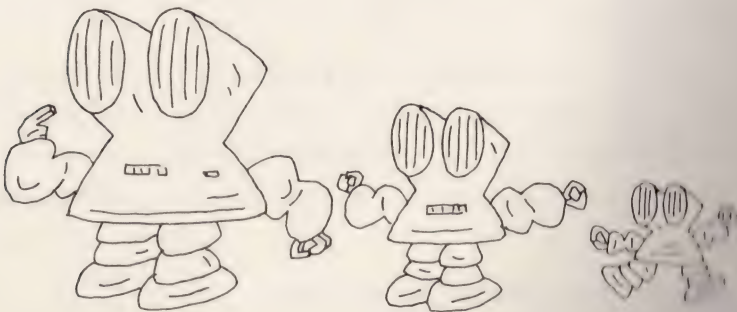



Figure-10.2.2　ASR命令の動作

ほとんどLSR命令の働きと同じですが，唯一違うのはビット7に0がはいるのではなく，もとのビット7の値がそのまま保存されることです．つまり符号を変えずに右シフトができるのです．図では＄A 2を1/2して＄D 1を得ていますが，10進数でいうと−94をシフトして−47になっており，正しく計算されていることがわかります．

アリスメティック・シフトのアドレッシングの例を次に示しておきます．使用できるアドレッシングモードは，ほかの単項演算命令と同じです．

ＡＳＲＡ………………………Ａレジスタの内容を1/2にする
ＡＳＲ　　＞＄●●●●……＄●●●●番地の内容を1/2にする

## 実習9　アリスメティック・シフト

8ビットのデータを2の補数表現されたものと考えて，ASR命令の実習を行います．メモリの1バイトデータに対してASR命令とLSR命令を実行し，その動作の違いを確認してみましょう．次のプログラムは，＄6830番地のデータを，符号なしのデータとして1/2したものを＄6831番地に，符号付きのデータとして1/2したものを＄6832番地に，それぞれストアするものです．スタート・アドレスは＄6800番地としてハンドアセンブルしています．

```
6800  B6 68 30  LDA  >$6830 ┐ 符号なしデータとし
6803  44        LSRA        │ て右シフトする
6804  B7 68 31  STA  >$6831 ┘ $6831番地にストアする
6807  B6 68 30  LDA  >$6830 ┐ 符号付きデータとし
680A  47        ASRA        │ て右シフトする
680B  B7 68 32  STA  >$6832 ┘ $6832番地にストアする
680E  3F        SWI
```

モニタでプログラムとデータを入力し，実行結果を確認してみましょう．

Figure-10.2.3　実習9(アリスメティック・シフト)の実行

```
*M6800↵……$6800番地からプログラムを入力する
6800 00-B6↵
6801 00-68↵
6802 00-30↵
6803 00-44↵
6804 00-B7↵
6805 00-68↵
6806 00-31↵
6807 00-B6↵
6808 00-68↵
6809 00-30↵
680A 00-47↵
680B 00-B7↵
680C 00-68↵
680D 00-32↵
680E 00-3F↵
680F 00-.↵

*M6830↵……$6830番地にシフトするデータを書き込む
6830 00-5A↵
6831 00-.↵
```

```
*D6800↵……プログラムとデータを確認する

6800 B6 68 30 44 B7 68 31 B6   ―プログラム
6808 68 30 47 B7 68 32 3F 00
6810 00 00 00 00 00 00 00 00
6818 00 00 00 00 00 00 00 00
6820 00 00 00 00 00 00 00 00
6828 00 00 00 00 00 00 00 00
6830 5A 00 00 00 00 00 00 00   ―データ
6838 00 00 00 00 00 00 00 00
*G6800↵……プログラムを実行する

*D6800↵……実行結果を確認する

6800 B6 68 30 44 B7 68 31 B6
6808 68 30 47 B7 68 32 3F 00   ―符号なし右シフト
6810 00 00 00 00 00 00 00 00
6818 00 00 00 00 00 00 00 00
6820 00 00 00 00 00 00 00 00
6828 00 00 00 00 00 00 00 00
6830 5A 2D 2D 00 00 00 00 00   ―符号付き右シフト
6838 00 00 00 00 00 00 00 00
*M6830↵……シフトするデータを変えてみる
6830 5A-C3↵
6831 2D-.↵

*G6800↵……もう一度実行する

*D6800↵……実行結果を確認する

6800 B6 68 30 44 B7 68 31 B6
6808 68 30 47 B7 68 32 3F 00   ―符号なし右シフト
6810 00 00 00 00 00 00 00 00
6818 00 00 00 00 00 00 00 00
6820 00 00 00 00 00 00 00 00
6828 00 00 00 00 00 00 00 00
6830 C3 61 E1 00 00 00 00 00   ―符号付き右シフト
6838 00 00 00 00 00 00 00 00
*
```



このプログラムの実行結果は＄6830番地のビット7が0のときは同じになり，1のときだけ違ってきます．10進数に直しながら結果を調べてみてください．また，もとのデータが奇数のときは1/2すれば当然1余りますから整数部分でしか正しく計算されません．余りがあるかないかはCフラグを見ればわかります*1．

*1 モニタのRコマンドでCCレジスタの内容を表示すれば，ビット0の状態でCフラグが1か0かを見ることができる

# 10・3 ローテート ROL, ROR

1バイトのデータのシフトは理解していただけたと思いますが，2バイト以上になるとそのようなシフト命令はないのでお手上げになってしまいます．そこで登場するのがROL(ROtate Left)，ROR(ROtate Right)という2つの命令です．これらは多バイト長の加減算をするときにADC命令，SBC命令が必要だったのと同じ理由で，多バイト長のシフトのために用意された命令です．Figure-10.3.1はローテート命令の動作の概念図です．

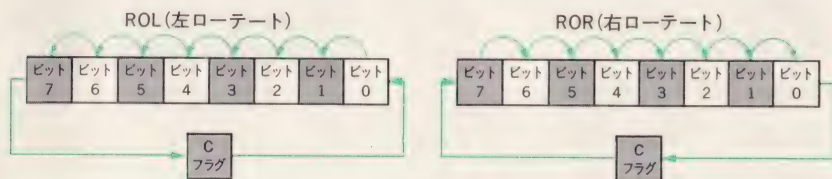



Figure-10.3.1 ローテート命令の動作

アキュムレータまたはメモリとCフラグとの9ビットで輪を作り，そのなかで左右にずらします．つまり9回ローテートを行えばもとの状態に戻ることになります．

Dレジスタを左にシフトする場合を例に，2バイトのデータのシフトを考えてみましょう．Figure-10.3.2を見てください．

まず，下位バイトであるBレジスタをLSL命令でシフトします．するとビット7の値はCフラグにはいりますが，これは本来上位バイトであるAレジスタのビット0にはいるべきものです．そこでROL命令でAレジスタを指定すれば，Aレジスタはシフトされるのと同時にCフラグの内容がビット0にはいります．こうして2バイトのシフトができるわけです．

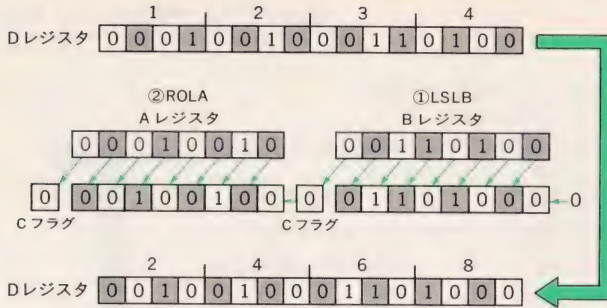



Figure-10.3.2　2バイトデータのシフト

ローテート命令は，シフトと同様にあふれたビットをCフラグに入れるので，ローテート命令を次々と実行すれば，何バイトのシフトでもできます．

### 実習10　多バイト長のシフト

LSL，ROL命令を使って多バイト長のシフトを行う実習です．

＄6930番地からの4バイトのデータを左に1つシフトするプログラムを作ってみましょう．スタート・アドレスは＄6900番地です．

```
6900   78 69 33   LSL  >$6933 …最下位バイトを左へシフトする（最上位ビットがCフラグにはいる）
6903   79 69 32   ROL  >$6932 ┐
6906   79 69 31   ROL  >$6931 ├ 下位バイトから順にCフラグを含めて左へローテートする
6909   79 69 30   ROL  >$6930 ┘
690C   3F         SWI
```

最下位バイトをシフトした後は，順々に上位バイトの方へ向かってローテートしていきます．

モニタでプログラムを入力して，実行結果を確認してみましょう．＄6930～＄6933番地にはシフトするデータをセットしておいてください．

Figure-10.3.3　実習10(多バイト長のシフト)の実行

```
*M6900↵………プログラムを入力する
6900 00-78↵
6901 00-69↵
6902 00-33↵
6903 00-79↵
6904 00-69↵
6905 00-32↵
6906 00-79↵
6907 00-69↵
6908 00-31↵
6909 00-79↵
690A 00-69↵
690B 00-30↵
690C 00-3F↵
690D 00-.↵

*M6930↵………シフトするデータ$1234ABCDを書き込む
6930 00-12↵
6931 00-34↵
6932 00-AB↵
6933 00-CD↵
6934 00-.↵

*D6900↵………プログラム，データを確認する

6900 78 69 33 79 69 32 79 69 ——プログラム
6908 31 79 69 30 3F 00 00 00
6910 00 00 00 00 00 00 00 00
6918 00 00 00 00 00 00 00 00
6920 00 00 00 00 00 00 00 00
6928 00 00 00 00 00 00 00 00
6930 12 34 AB CD 00 00 00 00 ——データ
6938 00 00 00 00 00 00 00 00
*G6900↵………プログラムを実行する

*D6900↵………実行結果を確認する

6900 78 69 33 79 69 32 79 69
6908 31 79 69 30 3F 00 00 00
6910 00 00 00 00 00 00 00 00
6918 00 00 00 00 00 00 00 00
6920 00 00 00 00 00 00 00 00
6928 00 00 00 00 00 00 00 00
6930 24 69 57 9A 00 00 00 00 ——左シフトした結果
6938 00 00 00 00 00 00 00 00
*
```

# 11

## 転送命令
## (レジスタ←→レジスタ)

●メモリ↔レジスタ間の転送には，LD，ST命令が用いられますが，レジスタ↔レジスタ間の転送には，TFR，EXG命令が用意されています．

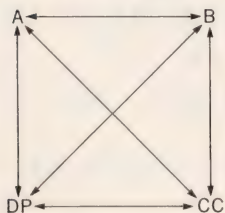
メモリ↔レジスタ間のアドレッシングモードが充実しているため，あまり利用価値がないようにも思われますが，DPレジスタへの値の代入やインデックス・レジスタにアキュムレータを代入する場合には，なくてはならない命令です．また，レジスタ↔レジスタ間の転送では，メモリのアドレスを指定する代わりにレジスタが用いられるため，このアドレッシング方式をレジスタモードと呼ぶこともあります．

# 11 トランスファ TFR

TFR(TransFeR)命令は，レジスタどうしでデータを転送をする命令です．レジスタ←→メモリ間の転送が自由自在に行えるので，TFR命令が使われるのは，主にDPレジスタに値をセットするときです．LD命令ではDPレジスタを扱えないため，アキュムレータなどを経由し，TFR命令でDPレジスタに値を入れます．また，インデックス・レジスタにアキュムレータの内容を入れる場合にも使われます．

6809のレジスタには8ビットのものと16ビットのものがありますが，TFR命令は8ビットは8ビットどうし，16ビットは16ビットどうしでしか意味を持ちません．Figure-11.1.1はTFR命令の動作を示したものですが，長さが等しいレジスタどうしならば，任意のレジスタ間でデータのコピーができます．

```
TFR    A, DP……………DPレジスタにAレジスタの内容をコピー
                    する(A → DP)
TFR    D, X………………Xレジスタに Dレジスタの内容をコピーす
                    る(D → X)
```

8ビットレジスタ間の転送

16ビットレジスタ間の転送

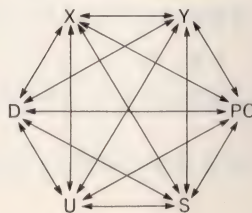



Figure-11.1.1　レジスタ間のデータ転送

いままでは各命令に何種類かのアドレッシングモードがあり，それと組み合せて1つの命令が形成できましたが，この命令は命令の性質上そのような概念はありません．その代わり，転送元，転送先のレジスタを指定するための1バイトが必要です．つまりTFR命令は常に2バイトの命令になります．巻末の命令表3を見ると，TFR命令は〝1 F●●〞とありますが，この〝●●〞の部分で2つのレジスタを指定します．すべてのレジスタには4ビットの番号が付けられており，転送元のレジスタ番号を上位4ビット，転送先のレジスタ番号を下位4ビットにあてはめ，その8ビットを〝●●〞の部分に置きます．各レジスタと番号はFigure-11.1.2に書いてあるように対応します．

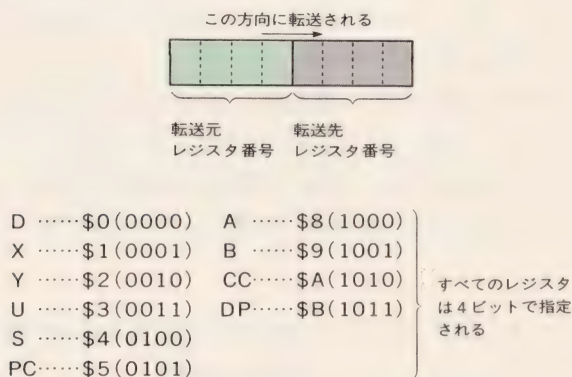



Figure-11.1.2 レジスタとレジスタ番号の対応

上であげた命令をハンドアセンブルしてみましょう．

```
1F  8B  ←――――――  TFR  A, DP
1F  9A  ←――――――  TFR  B, CC
1F  01  ←――――――  TFR  D, X
```

なお，TFR命令の実習は，次節のEXG命令と合せて行います．

# 11.2 エクスチェンジ EXG

EXG(EXchanGe)命令は2つのレジスタの内容を交換します．ちょうどBASICのSWAP文のような命令です．TFR命令が一方のレジスタの内容をもう一方のレジスタにコピーするだけなのに対し，この命令は両方のレジスタが変化するところが異なっています．Figure-11.2.1は，EXG命令の動作を示したものです．

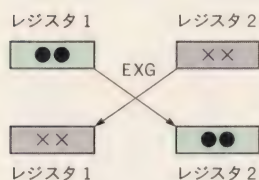
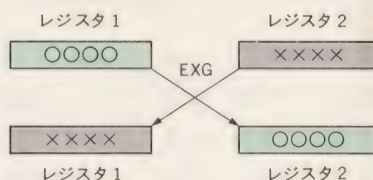



Figure-11.2.1　EXG命令の動作

EXG命令で指定する2つのレジスタにはTFR命令のような転送元，転送先といった区別はないので，どちらを左または右に書くかは問題ではありません．EXG命令の使用例とそのアセンブル結果は次のとおりです．

```
1E  89  EXG   A, B……Aレジスタと B レジスタの内容を入れ替える
1E  01  EXG   D, X……Dレジスタと X レジスタの内容を入れ替える
```

## 実習11　レジスタ↔レジスタ間のデータ転送

TFR, EXG 命令の実習を行います．次のプログラムを読んで，それを実行したときメモリがどのように変化するか予想してください．

```
LDD   >$6A30
EXG   A, B
STD   >$6A32
TFR   A, B
STD   >$6A34
SWI
```

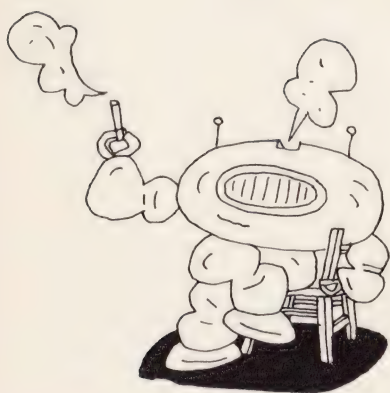
＄６Ａ３３番地が＄６Ａ３０番地の内容と，＄６Ａ３２，＄６Ａ３４，＄６Ａ３５番地が＄６Ａ３１番地の内容とそれぞれ等しくなります．

ハンドアセンブルした結果は次のようになります．

```
6A00   FC 6A 30   LDD   >$6A30
6A03   1E 89      EXG   A, B
6A05   FD 6A 32   STD   >$6A32
6A08   1F 89      TFR   A, B
6A0A   FD 6A 34   STD   >$6A34
6A0D   3F         SWI
```

プログラムとデータをセットして，実行結果を確認してみましょう．

Figure-11.2.2 実習11(レジスタ↔レジスタ間のデータ転送)の実行

```
*M6A00↲……プログラムを入力する
6A00 00-FC↲
6A01 00-6A↲
6A02 00-30↲
6A03 00-1E↲
6A04 00-89↲
6A05 00-FD↲
6A06 00-6A↲
6A07 00-32↲
6A08 00-1F↲
6A09 00-89↲
6A0A 00-FD↲
6A0B 00-6A↲
6A0C 00-34↲
6A0D 00-3F↲
6A0E 00-.↲

*M6A30↲……転送するデータを入力する
6A30 00-F5↲
6A31 00-01↲
6A32 00-.↲

*D6A00↲……プログラム，データを確認する

6A00 FC 6A 30 1E 89 FD 6A 32 ――プログラム
6A08 1F 89 FD 6A 34 3F 00 00
6A10 00 00 00 00 00 00 00 00
6A18 00 00 00 00 00 00 00 00
6A20 00 00 00 00 00 00 00 00
6A28 00 00 00 00 00 00 00 00
6A30 F5 01 00 00 00 00 00 00 ――データ
6A38 00 00 00 00 00 00 00 00
*G6A00↲

*D6A00↲……実行結果を確認する

6A00 FC 6A 30 1E 89 FD 6A 32
6A08 1F 89 FD 6A 34 3F 00 00
6A10 00 00 00 00 00 00 00 00
6A18 00 00 00 00 00 00 00 00
6A20 00 00 00 00 00 00 00 00
6A28 00 00 00 00 00 00 00 00
6A30 F5 01 01 F5 01 01 ――データ転送の結果
6A38 00 00 00 00 00 00 00 00
*
```

# 12 スタックを扱う命令

●コンピュータのデータ構造の1つにスタック(Stack)というものがあります．これはLIFO(Last In First Out)とも呼ばれる基本的かつ重要なデータ構造であり，コンピュータの世界では広く応用されています．ここではスタックについての基本的な概念とその使われ方の一部を紹介します．

# 12・1 スタックの概念

プログラムを組んでいくと，レジスタに必要なデータがはいっているにもかかわらず，そのレジスタを使いたい場合があります．たとえばA，Bレジスタとも大事なデータがはいっているのに，加算をしなければならないときなどがそうです．もしそのまま加算をすれば，アキュムレータの内容は失われてしまいます．できればこのような事態は避けたいのですが，いつも避けられるとは限りません．

このような場合は，いったんアキュムレータの内容をメモリにストアしてから演算を行い，その後でもう一度アキュムレータに戻してやるという方法が考えられます．ところが大きなプログラムになると，こんなことが1か所や2か所では済まなくなり，そのたびに別々のアドレスにしまっていては必要なメモリの量もバカになりません．それではAレジスタは何番地，Bレジスタは何番地というようにあらかじめ各レジスタをストアするアドレスを決めておけばよいかというとそうもいきません．このことはFigure-12.1.1を見ればなぜ不都合なのかわかると思います．

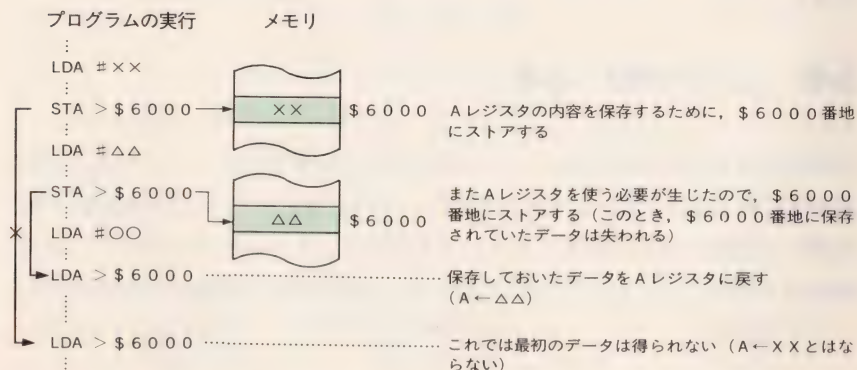



Figure-12.1.1　特定アドレスにレジスタを退避する場合

それではどうしたらこれらの処理が矛盾なく行えるのでしょうか．それに対する実に明快な答えとして，スタックを利用するという方法があります．

**スタック**とは〝積む〟とか〝(物を積んだ)山〟の意味ですが，コンピュータの世界でいうスタックは，データを積み重ねた形の記憶装置です．Figure-12.1.2はさきほどの処理をスタックを用いて行った様子を示しています．

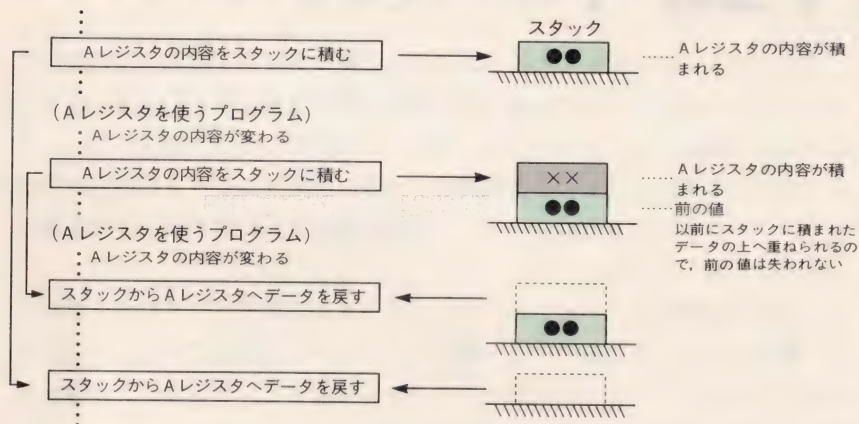



Figure-12.1.2　スタックを利用したレジスタ退避

このように，スタックはデータを順々に積み重ねていき，必要に応じて上から取り出せるようになっています．つまり，最後にスタックへ入れたデータは最初に取り出すことになるので，LIFO(Last In First Out)と呼ばれるのです．

## スタックの構造

実際のコンピュータでは，スタックは特別に用意されたハードウェアを利用するのではなく，Sレジスタ(システムスタック・ポインタ)とメモリの一部を使って実現されています．SレジスタはプログラムカウンタやXレジスタなどと同様，16ビットのレジスタでアドレス空間の任意のアドレスを指すことが可能です．

Figure-12.1.3の，データをスタックにしまったり，スタックからデータを取り出したりしたときのSレジスタの動きに注意してください．データをスタックにしまうとき，**まずSレジスタをデクリメント**して，次にSレジスタの指すアドレスにデータが格納されます．また，スタックからデータを取り出すときは，Sレジスタの指すアドレスの内容をデータとして**取り出してから，Sレジスタをインクリメント**します．つまり，いつもSレジスタは最後にしまったデータのあるアドレスを指すようにするのです．

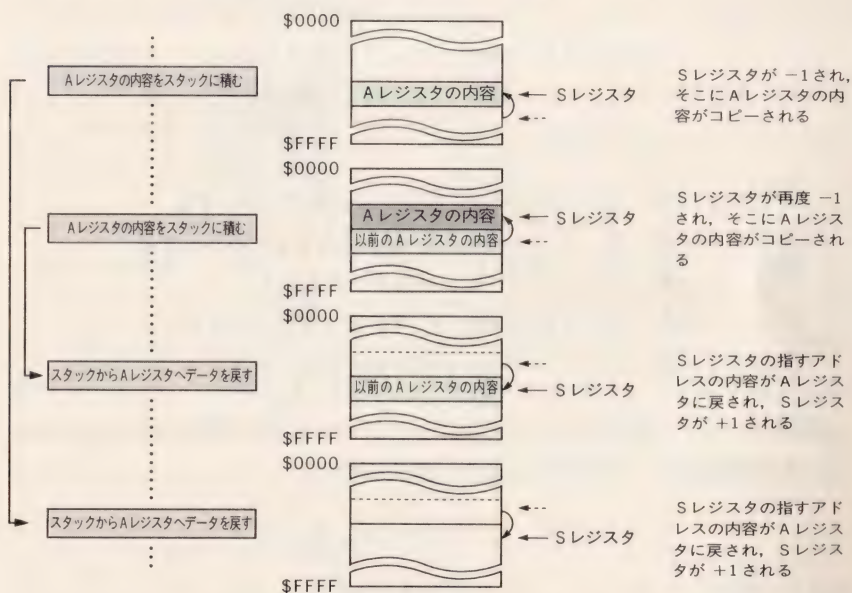



Figure-12.1.3　スタックとSレジスタの動作

このようにしてスタックは実現できるのですが，実際にはこれらの一連の動作(Sレジスタをデクリメントしてからデータをしまう等)は，1つの命令で自動的に行うことができるので，プログラムを書く場合はSレジスタの値(データがしまわれるアドレス)やSレジスタをデクリメントするといったことは意識する必要はありません．これらの動作を行わせる命令が，次に解説するPSH(PuSH（プッシュ）)／PUL(PULl（プル）)命令なのです．

# 12・2 スタックを利用するPSH, PUL

レジスタの内容をスタックにしまうことをプッシュ(push)，取り出すことをプル(pull)といいます．プッシュ／プルは任意のレジスタに対してできますが，Sレジスタだけは，それ自身がスタックを管理するレジスタなのでできません．また次に示すように，6809では同時に複数のレジスタをプッシュまたはプルすることができます．

PSHS　Y，U……………Y，Uレジスタをプッシュする
PSHS　A，B，X………A，B，Xレジスタをプッシュする
PULS　A…………………Aレジスタにプルする
PULS　B，X，Y………B，X，Yレジスタにプルする
PULS　U…………………Uレジスタにプルする

複数のレジスタのプッシュ／プルは各レジスタに優先順位が付いているので，その順番に従って実行されます．優先順位は，

PC，U，Y，X，DP，B，A，CC

のように決まっており，プッシュするときはPCからCCレジスタ，プルするときはCCレジスタからPCの順で行われます．上の例を実行すると，スタックはFigure-12.2.1のように動きます．

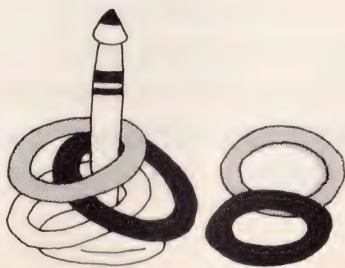
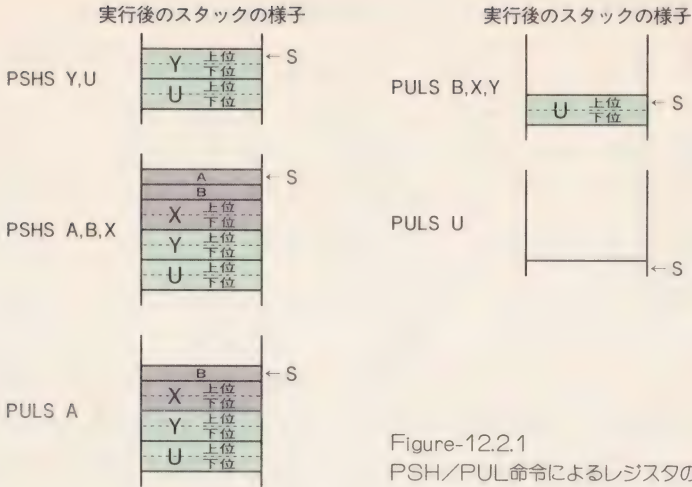



Figure-12.2.1
PSH／PUL命令によるレジスタの退避

巻末の命令表4を見ると，PSHSは〝34●●〟，PULSは〝35●●〟となっていますが，この〝●●〟のところでレジスタを指定します．プッシュ／プルできるレジスタはちょうど8つなので，各ビットにレジスタを1つずつ割り当てます．この対応はプッシュ，プルともに共通で，命令表4の下に記載してあります．そしてFigure-12.2.2に示すように，対象となるレジスタに対応するビットを1にした1バイトを，$34または$35に続けて与えます．

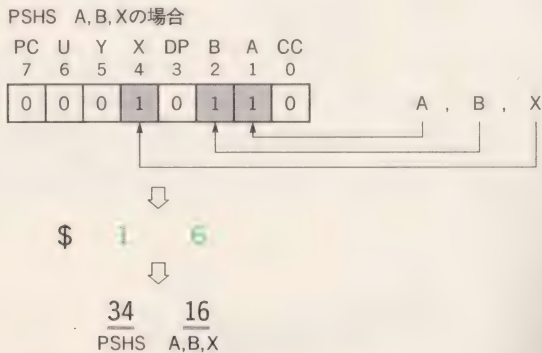



Figure-12.2.2 PSH／PUL命令のレジスタ指定と4ビットの対応

さきほどの例をハンドアセンブルすると次のようになります.

```
34 60 ←———— PSHS Y, U
34 16 ←———— PSHS A, B, X
35 02 ←———— PULS A
35 34 ←———— PULS B, X, Y
35 40 ←———— PULS U
```

6809には,スタック・ポインタと呼ばれるレジスタが2つ(SレジスタとUレジスタ)あり,両者は独立して機能することができます*1.Uレジスタに対するプッシュ/プルもSレジスタとまったく同様に行えます.ただし今度はUレジスタがスタックを管理しているので,Uレジスタの代わりにSレジスタをプッシュ/プルすることができます.レジスタを指定するビットパターンはUレジスタのビット(ビット6)がSレジスタになるだけでほかは同じになっています.

このようにSレジスタとUレジスタは対等の機能を持っていますが,サブルーチンの呼び出しや割込みでスタックが使われる場合には,常にSレジスタが使われます.

###  実習12 スタックを使う

システムスタックにレジスタの内容を保存するプログラムを作り,PSH,PUL命令の実習を行います.次のプログラムでは,最初にAレジスタにロードしたデータをいったんシステムスタックにプッシュして,その後Aレジスタに加えた値をメモリにストアします.さらに,加算後システムスタックからAレジスタにプルしたデータをメモリにストアしています.Sレジスタはコンピュータが起動したときセットされるので,ここではあえてセットし直す必要はありません*2.

*1 Sレジスタが管理するスタックをシステムスタック,Uレジスタが管理するスタックをユーザースタックと呼ぶ.本書で扱っているのはシステムスタックのみ
*2 SレジスタはBASICインタープリタによってあらかじめセットされる

ハンドアセンブルした結果を次に示します．スタート・アドレスは＄６Ｂ００番地です．

```
6B00    86 05          LDA   #5
6B02    34 02          PSHS  A  ………………Aレジスタの内容を
                                          スタックにプッシュ
                                          する
6B04    8B 03          ADDA  #3
6B06    B7 6B 30       STA   >$6B30
6B09    35 02          PULS  A  ………………スタックにプッシュ
                                          したデータをAレジ
                                          スタにプルする
                                          (A←5)
6B0B    B7 6B 31       STA   >$6B31
6B0E    3F             SWI
```

プログラムを実行してみると，＄６Ｂ３１番地にストアされるデータは，最初にＡレジスタにセットされた値がそのままはいり，Ａレジスタの内容が保存されたことがわかります．この実習では単にＡレジスタの値を退避しただけですが，他のレジスタについてもいろいろと試してください．

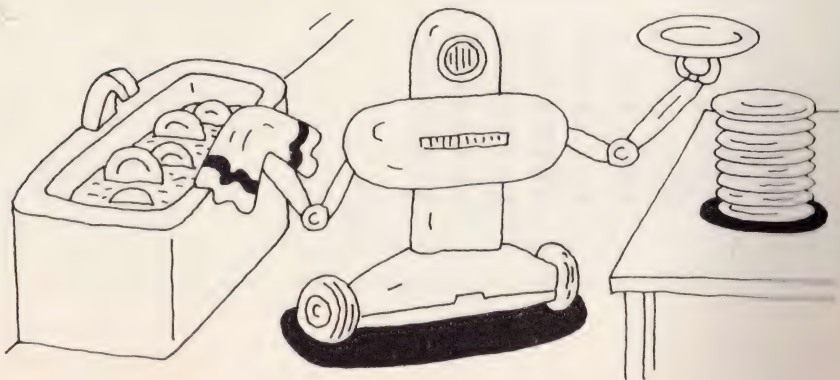
また，スタックはレジスタの内容を退避するためだけでなく，プログラムを実行する際のワークエリアなどにも利用されます(クイックソートなど)．本書では，スタックの利用方法について解説していないので，興味のある方はプログラミングの参考書などを調べてみるのもよいでしょう．

モニタでプログラムを入力し，実行結果を確認してみましょう．

Figure-12.2.3　実習12(スタックを使う)の実行

```
*M6B00↲……プログラムを入力する
6B00 00-86↲
6B01 00-05↲
6B02 00-34↲
6B03 00-02↲
6B04 00-8B↲
6B05 00-03↲
6B06 00-B7↲
6B07 00-6B↲
6B08 00-30↲
6B09 00-35↲
6B0A 00-02↲
6B0B 00-B7↲
6B0C 00-6B↲
6B0D 00-31↲
6B0E 00-3F↲
6B0F 00-.↲

*D6B00↲――プログラムを確認する

6B00 86 05 34 02 8B 03 B7 6B ――プログラム
6B08 30 35 02 B7 6B 31 3F 00
6B10 00 00 00 00 00 00 00 00
6B18 00 00 00 00 00 00 00 00
6B20 00 00 00 00 00 00 00 00
6B28 00 00 00 00 00 00 00 00
6B30 00 00 00 00 00 00 00 00
6B38 00 00 00 00 00 00 00 00
*G6B00↲――プログラムを実行する

*D6B00↲――実行結果を確認する

6B00 86 05 34 02 8B 03 B7 6B
6B08 30 35 02 B7 6B 31 3F 00
6B10 00 00 00 00 00 00 00 00
6B18 00 00 00 00 00 00 00 00
6B20 00 00 00 00 00 00 00 00
6B28 00 00 00 00 00 00 00 00
6B30 08 05 00 00 00 00 00 00
6B38 00 00 00 00 00 00 00 00
*
```

- 05：スタックに保存したデータ(もとの値)
- 08：ADD命令で3を足したデータ(5＋3)

# 13 分岐命令

●マシン語のプログラムは，アドレスの低い方($0000番地）から高い方（$FFFF番地）へ向かって順にメモリ上に並んでおり，CPU は PC（プログラム・カウンタ）を使って，それらの命令を遂次読み出しては解析／実行していきます．つまり，プログラム・カウンタを強制的に変更しない限り流れはアドレスの低い方から高い方へ向かって実行されるだけです．

しかし，これだけではほとんどのプログラムは作ることはできないでしょう．条件判断によって別のプログラムを実行したり，サブルーチンを呼んだりできなくては，自由にプログラムを書くことなど不可能です．本章では，このプログラムの流れを変える，分岐命令について解説します．

# 13・i プログラムの流れを変えるJMP（絶対アドレス指定の分岐命令）

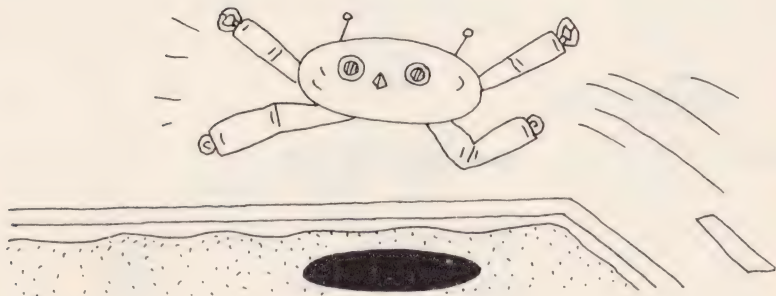
JMP（JuMP）命令はBASICのGOTO文にあたり，無条件にプログラム・カウンタの値をセットし直す命令です．プログラムが，あるアドレスまで実行されたときに$●●●●番地へ分岐したいのであれば，そこに

JMP　＞$●●●●……$●●●●番地へ分岐する

と書けば，CPUはこの命令を実行すると，ただちに$●●●●をプログラム・カウンタに取り込み，次から実行する命令は$●●●●番地からになります．

JMP命令にも3種類のアドレッシングモードが用意されており，上に示したエクステンドのほかに，ダイレクトとインデックスが使えます．

JMP　＜$●●…………DPレジスタの値を上位バイトとして$●●番地へ分岐する

JMP　，X………………Xレジスタの指すアドレスへ分岐する（X→PC）

エクステンドモードを例に，JMP命令の動作の様子をFigure-13.1.1に示しておきましょう．

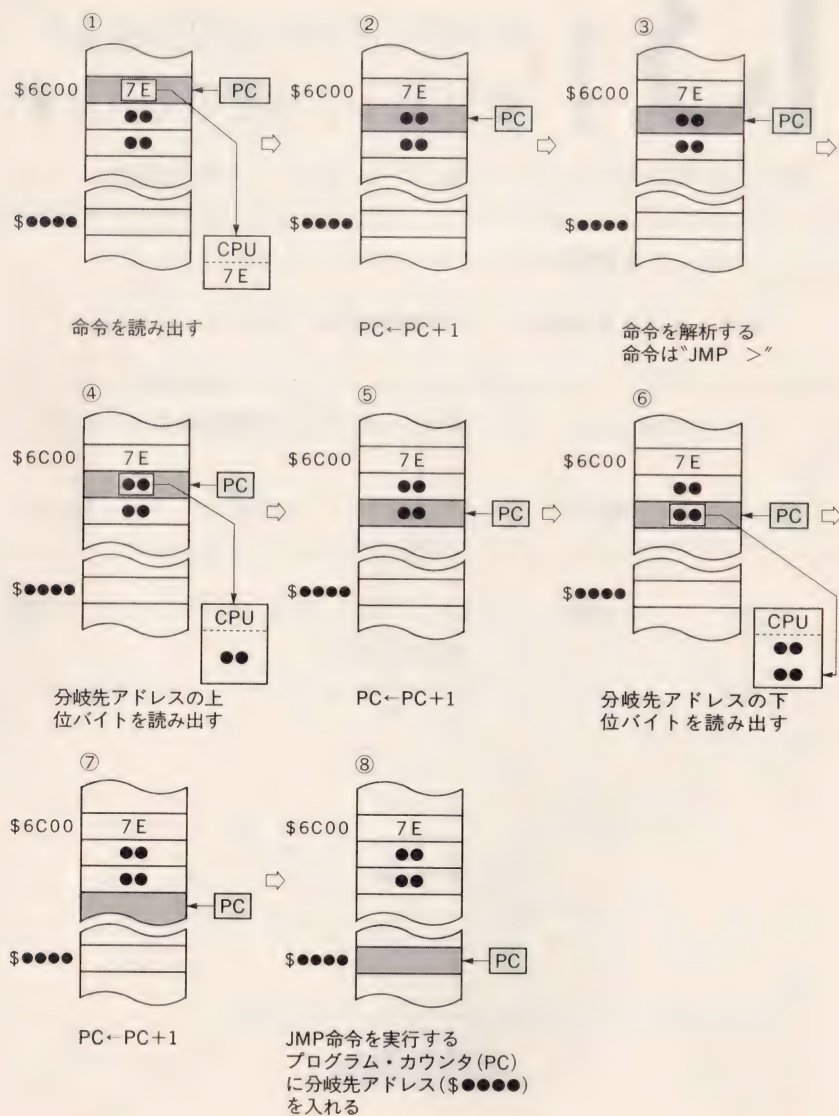



Figure-13.1.1　JMP命令の動作

# 13・2 サブルーチンを呼ぶ JSR/RTS

JSR(Jump to SubRoutine)命令はサブルーチンを呼ぶ命令で，ちょうどBASICのGOSUB文にあたります．JMP命令と同様，エクステンド，ダイレクト，インデックスの各アドレッシングモードにより，分岐先のアドレスを指定できます．

JSR命令がJMP命令と違う点は，分岐先(サブルーチン)でRTS(ReTurn from Subroutine)命令を実行すると，JSR命令を行った次の命令のアドレスへ戻ってくることです．このことはBASICのGOSUB～RETURNとまったく同じです．

では，なぜRTS命令によってサブルーチンから戻ることができるのでしょうか．そのためには，戻るべきアドレス(つまりJSR命令を実行したときのプログラム・カウンタの値)をどこかにしまっておかなくてはなりません．ところが，このアドレスを特定のレジスタやメモリにしまってサブルーチンを呼んだのでは，サブルーチンのなかでさらにほかのサブルーチンが呼ばれた場合に，最初のアドレスは失われてしまうのです．

実は，この議論は前章のPSH，PUL命令とまったく同じように，スタックを使うことによって解決できます．Figure-13.2.1はJSR／RTS命令の実行される様子です．

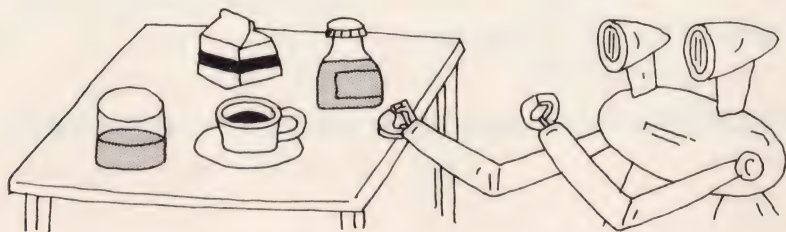



Figure-13.2.1　JSR／RTS命令の実行

CPU が JSR 命令を取り込むと(このときプログラム・カウンタはすでに次の命令のアドレスを指している．5.2 参照)，まずプログラム・カウンタの内容をスタックにプッシュします．つまりこれが RTS 命令が実行されたときに戻り番地になるのです．この場合のスタックとは，システムスタック（S）のことです．次に分岐先のアドレスをプログラム・カウンタに入れて，サブルーチンの呼び出しは完了します．JSR 命令はこの 2 つの動作を自動的に行ってくれます．

また，CPU はサブルーチンのなかで RTS 命令を取り込むと，戻り番地としてスタックからアドレスを取り出し，それをプログラム・カウンタに入れてサブルーチンからメインルーチンに戻るのです．

この原理を応用すれば，サブルーチンのなかでさらにサブルーチンが呼ばれても正常に動作することができます．Figure-13.2.2 がその例を示したものですが，＄６０２０番地で＄６１００番地のサブルーチンを呼び，そのサブルーチンのなかで，また＄６１５０番地を呼んでいます．最初の JSR 命令では，スタックにはいる戻り番地は＄６０２３ですが，２回目の JSR 命令のときは，スタックには＄６１２Bがはいります．このとき前の戻り番地（＄６０２３）は消えずにその上に積み上げられるため，＄６１５０番地からのサブルーチンから戻ったとき，最初に JSR 命令を実行したルーチンへの戻り番地が残っているのです．

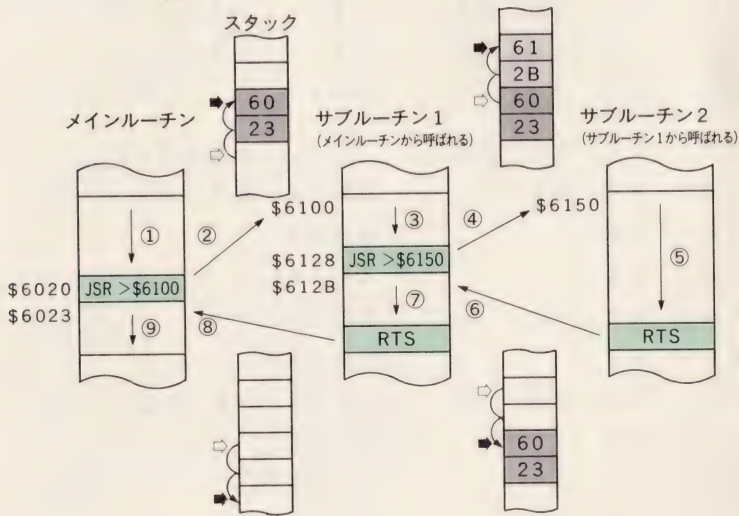



Figure-13.2.2　サブルーチンのネスト

このようにスタックの上には戻り番地が順番に積まれていくため，何重にでもサブルーチンを呼び出すことができます．

##  実習13 サブルーチンの利用

メインルーチンからサブルーチンを呼び出すプログラムを作り，JSR／RTS 命令の実習を行います．

次の＄６Ｃ００番地からのプログラムは，＄６Ｃ１８番地からのサブルーチンを３回呼んで，そのたびにＤレジスタの内容をメモリにストアするものです．サブルーチンは 16 ビットの乱数を発生するもので，乱数のデータをＤレジスタに持ってメインルーチンへ戻ります．

```
6C00  BD 6C 18   JSR   >$6C18  ┐
6C03  FD 6C 38   STD   >$6C38  │ (メインルーチン)
6C06  BD 6C 18   JSR   >$6C18  │ サブルーチンを３回
6C09  FD 6C 3A   STD   >$6C3A  │ 呼んで，それぞれの
6C0C  BD 6C 18   JSR   >$6C18  │ 結果を＄６Ｃ３８～
6C0F  FD 6C 3C   STD   >$6C3C  ┘ ＄６Ｃ３Ｄ番地にストアする
6C12  3F         SWI

6C18  FC 6C 2E   LDD   >$6C2E  ┐
6C1B  84 08      ANDA  #$08    │
6C1D  48         ASLA          │
6C1E  48         ASLA          │
6C1F  48         ASLA          │
6C20  48         ASLA          │ (サブルーチン)
6C21  B8 6C 2E   EORA  >$6C2E  │ 乱数を発生するサブルーチン
6C24  58         ASLB          │
6C25  49         ROLA          │
6C26  C9 00      ADCB  #0      │
6C28  FD 6C 2E   STD   >$6C2E  │
6C2B  39         RTS           ┘

6C2E  [7D 53] ………………乱数のたね
```

このサブルーチンについての詳しい説明はしませんが，使われている命令はどれもすでに出てきたものばかりです．興味のある人はどのように動作しているのか調べてみるのもよい勉強になると思います．

プログラムをモニタで入力し，実行結果を確認してみましょう．

Figure-13.2.3 実習13(サブルーチンの利用)の実行

```
*M6C00↵………プログラム(メインルーチン)を入力する
6C00 00-BD↵
6C01 00-6C↵
6C02 00-18
  :
6C11 00-3C↵
6C12 00-3F↵
6C13 00-.↵

*M6C18↵………プログラム(サブルーチン)を入力する
6C18 00-FC↵
6C19 00-6C↵
  :
6C2A 00-2E↵
6C2B 00-39↵
6C2C 00-.↵

*M6C2E↵………乱数のたねをセットする
6C2E 00-7D↵
6C2F 00-53↵
6C30 00-.↵

*D6C00↵………プログラム，データを確認する

6C00 BD 6C 18 FD 6C 38 BD 6C  ── メインルーチン部
6C08 18 FD 6C 3A BD 6C 18 FD
6C10 6C 3C 3F 00 00 00 00 00
6C18 FC 6C 2E 84 08 48 48 48  ── サブルーチン部
6C20 48 B8 6C 2E 58 49 C9 00
6C28 FD 6C 2E 39 00 00 7D 53  ── 乱数のたね
6C30 00 00 00 00 00 00 00 00
6C38 00 00 00 00 00 00 00 00
*G6C00↵

*D6C00↵

6C00 BD 6C 18 FD 6C 38 BD 6C
6C08 18 FD 6C 3A BD 6C 18 FD
6C10 6C 3C 3F 00 00 00 00 00
6C18 FC 6C 2E 84 08 48 48 48
6C20 48 B8 6C 2E 58 49 C9 00
6C28 FD 6C 2E 39 00 00 EA 9D  ── 3回目のサブルーチンの呼び出しで
6C30 00 00 00 00 00 00 00 00     発生した乱数のたね
6C38 FA A7 F5 4E EA 9D 00 00
*                              ── メモリに書き込んだ乱数
```

# 13・3 条件判断とブランチ命令B○○(相対アドレス指定の分岐命令)

ブランチ命令は**相対アドレス指定**の分岐命令のことで，JMP 命令などと同様プログラムの流れを変える命令です．ただ分岐先のアドレスの与え方がJMP 命令とは異なります．

ブランチ命令は JMP 命令のように〝何番地へ分岐せよ″という形ではなく，〝何番地先(または後)へ分岐しろ″という形で表します．つまりブランチ命令が書いてあるアドレスに対して，相対的に分岐先を指定するのです(リラティブモード)*1.

この違いを BRA(BRanch Always)命令と JMP 命令を例に解説しましょう．Figure-13.3.1 は＄６０１０番地から＄６０１８番地へ分岐する命令を，BRA 命令と JMP 命令それぞれの場合について書いたものです．

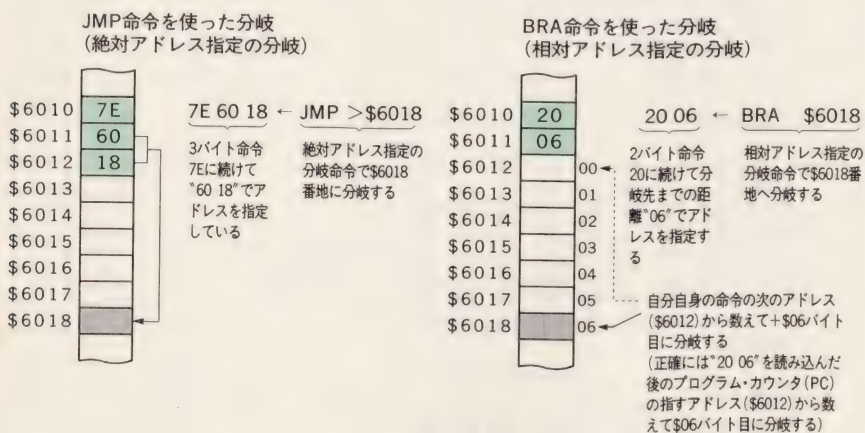



Figure-13.3.1 相対アドレス指定と絶対アドレス指定の分岐の違い

*1 ブランチ命令のアドレッシングは，すべてリラティブモードに分類される．Figure-6.2.1 参照

JMP 命令はエクステンドモードを使うので，アセンブルすると＄７Ｅになり，BRA 命令は巻末の命令表６から＄２０であることがわかります．JMP 命令の場合は＄７Ｅに続けて分岐先のアドレスを〝６０　１８″のように指定しますが，BRA 命令の場合は，＄６０１８番地までの距離(バイト数)を１バイトで表し，それを〝２０″に続けて書きます．距離の数え方は〝２０●●″の次のアドレス，すなわち＄６０１２番地を０として数えます．数えてみると距離は６(＄０６)ですので，この場合は〝２０　０６″となります．これはアドレスの高い方への分岐ですが，低いアドレスへの分岐の場合は分岐先までの距離がマイナスになります．例えば上の例で＄６０００番地へ分岐しようとすれば，−18 になりますが 16 進数でどう書けばよいのかちょっと迷うところです．実は，〝２０″に続けて書く距離を表す１バイトは２の補数表現の値なので，＄６０００までの距離，すなわち−18 は＄ＥＥで表され，〝２０ＥＥ″で＄６０００番地へ分岐できます．結局ブランチ命令で分岐することのできるアドレスは−128〜＋127(＄８０〜＄７Ｆ)の範囲になります．

またこれより遠いアドレスへは分岐できないかというと，6809 には，64 K バイトのアドレス空間のどこへでも分岐することができる**ロングブランチ命令**[*1]が用意されています．BRA 命令の代わりに LBRA 命令を使えば 16 ビットで距離を指定することができるのです(Figure-13.3.2)．

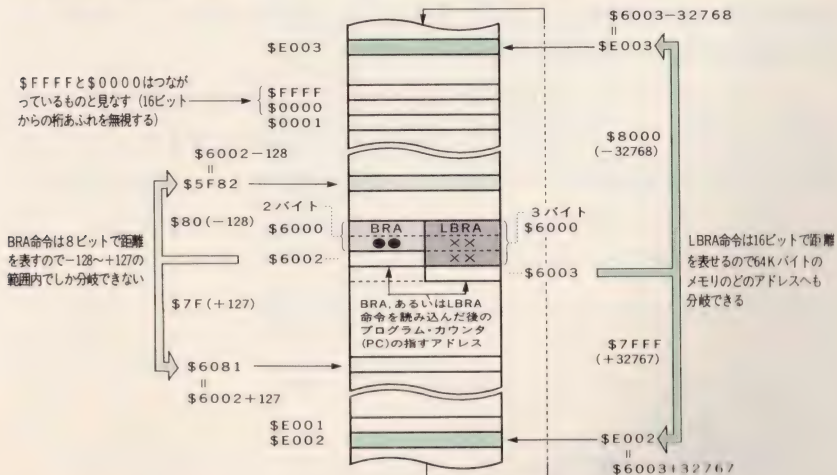



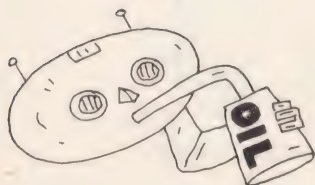
Figure-13.3.2　ブランチ命令とロングブランチ命令の分岐範囲

*1 すべてのブランチ命令には，対応するロングブランチ命令がある．ニーモニックは，ブランチ命令の前に〝L″を付けて表す．巻末命令表６参照

## “リロケータブル(再配置可能)なプログラム”

では，ブランチ命令を使う場合とジャンプ命令を使う場合には，それぞれどのような利点，欠点があるのでしょうか．確かにFigure-13.3.1の例ではジャンプ命令は3バイト，ブランチ命令は2バイトなので，ブランチ命令の方が短くて済み経済的です．しかしロングブランチ命令となると3バイト必要なので，ジャンプ命令と変わらなくなってしまい，面倒なアドレス計算をするブランチ命令の方が不便なような気がします．ところがジャンプ命令とブランチ命令とでは，根本的に異なることがあるのです．

いま仮に，実習13のプログラムを＄6D00番地へ移したとすると，果たしてそのプログラムは動くでしょうか．答えはNOです．なぜならば，このプログラムはJSR命令を使って絶対アドレスを指定しているため，“JSR　>＄6C18”はそのままの形で残ってしまい，JSR命令の分岐先を変更して，“JSR　>＄6D18”としなければプログラムは動きません．では，もしサブルーチンの呼び出しをブランチ命令で行った場合にはどうなるでしょうか．ブランチ命令は絶対的なアドレスで指定するのではなく，距離(相対的なアドレス)で指定するので，プログラム全体の格納されるアドレスが変わっても，プログラム内部の相対的な位置関係は変わらないため，正常に動作させることが可能です．このようなプログラムの性質を，**リロケータブル**(再配置可能)であるといいます．

このようにプログラム内の分岐には，ジャンプ命令を使うよりブランチ命令を使った方が，より汎用的なプログラムにすることができます．また，次に解説する条件分岐命令はすべてブランチ命令(相対アドレスによる指定)となっており，ジャンプ命令(絶対アドレスによる指定)にはありません．

# 13・4 フラグと条件分岐

マシン語での条件判断は，ある条件のときには分岐し，その他のときは分岐しない，という形で行われます．条件とは，一方より他方が大きいとか小さいとか，等しいなどのことを指します．そしてそれらの情報はCCレジスタの各種フラグが0か1かによって与えられる情報なのです．

CCレジスタと各フラグの機能をFigure-13.4.1に示しておきます．CCレジスタのフラグは全部で8つありますが，本書で扱っているのはC，Z，Nフラグの3つです．

なお，各種演算命令が実行されたときのC，Z，Nフラグ変化の様子は巻末の命令表に示されています．

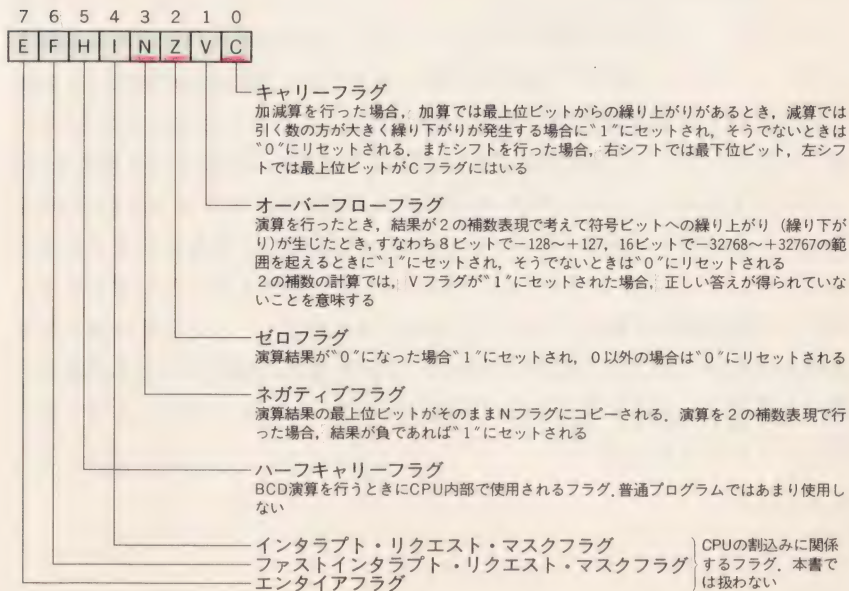



Figure-13.4.1 CCレジスタと各フラグの機能

8章，10章で解説したフラグは，加減算やシフトのところで出てきたCフラグだけですが，条件分岐の判断の材料として使われるフラグには，Z，Nなどのフラグもあります．これらのフラグは主に命令表1,2の命令を実行したときに，その結果に応じてセット／リセットされるものです．

## “比較と条件分岐”

8章で解説した命令のなかにCMP命令がありましたが，これはまさにここで説明する条件分岐のための命令なのです．CMP命令はSUB命令と同様に減算を行う命令ですが，この命令を実行すると，C，Z，Nの3つのフラグはすべて変化します．CフラグはSUB命令のところで述べたように，演算結果に繰り上がり／繰り下がりがあったときにセットされますが，Zフラグは演算結果が0のときセット(1になる)され，Nフラグは演算結果が負のときセットされるので，このような名前が付いているのです．

ではCMP命令に注目して，フラグの変化の意味と，それを利用しての判断について解説を行いましょう．

いま，Aレジスタの内容が10以上のときと10未満のときとで別の処理をしたいとします．BASICであれば〝IF　A>=10　THEN～ELSE～〟のように書くことができますが，マシン語ではこうはできません．まず〝A>=10〟がどのようにしたら得られるかが問題です．これは次のように考えます．

〝A>=10〟は〝A−10>=0〟と同じことですから，Aレジスタから10を引いたときに繰り下がりがなく，Cフラグが立たなければAレジスタの内容は10以上，立てばAレジスタの内容は10未満と判断することができます．そこでCMP命令を使ってAレジスタから10を引いて，フラグを変えさせればよいのです．そしてCMP命令の次にCフラグの状態によって分岐する命令を書けば，このような処理が実現できます．これをプログラムにすると次のようになります．

```
         CMPA   #10 ……………Aレジスタの内容から10を引い
                            てフラグを変える
         BCS    MIMAN………Cフラグが1ならMIMANへ分
         (10以上のときの処理)   岐せよ
                 ⋮
MIMAN    (10未満のときの処理)
                 ⋮
```

ここに出てきた〝MIMAN〟とは**ラベル**というものです．ラベルは，アドレスなどに付ける名前で，MIMANは10未満のときの処理をするプログラムに筆者が勝手に付けた名前です．実際にアセンブルすればラベルは1つのアドレスを意味するのですが，プログラムを組むときは，ラベルの付けられた処理ルーチンのアドレスが何番地になるかなどということはまったく意識する必要はないので，このようにラベルを付けて分岐先を表しておくのです．

このプログラムでは，条件分岐命令としてBCS(Branch Carry Set)(ブランチ キャリー セット)命令が使われていますが，BCS命令はCフラグが1のときに処理ルーチンMIMANに分岐し，0のときはその下の命令を実行します．こうして条件判断をして別々の処理をすることができるのです．

またBCS命令のところをBEQ(Branch EQual)(ブランチ イコール)命令やBMI(Branch MInus)(ブランチ マイナス)命令にすれば，それぞれAレジスタが10と等しいとき，10を引いてマイナス(＄８０～＄ＦＦの範囲)になるときなどに，分岐ができるようになります．これ以外の分岐命令の条件や動作については巻末の命令表6を参照してください．

条件判断で大切なことは，フラグの変わる命令とフラグの状態によって分岐する命令の両方が常に必要であることです．そのためいろいろな命令を実行したとき，各フラグがどのようにセットされるのかを命令表を見てよく理解しておかなければなりません．

##  実習14　条件分岐

条件分岐命令の実習として，＄６Ｄ３０番地にはいっているデータの１になっているビットの数を数えて，その数を＄６Ｄ３１番地にストアするプログラムを書いてみましょう．このプログラムではＢレジスタをループ・カウンタとして使い，ループを構成していることに注意してください．

```
       LDA   >$6D30……$6D30番地の内容をAレジ
                      スタにロードする
       CLR   >$6D31……$6D31番地の内容を0にする
       LDB   #8 ………… 8回をカウントするループ・カウ
                      ンタにBレジスタを使う
LOOP   ASLA…………………Aレジスタの内容を1つ左へシ
                      フトしてCフラグを変える
       BCC   ZERO……… Cフラグが0ならZEROへ分岐
                      する
       INC   >$6D31……$6D31番地の内容をインクリ
                      メント(+1)する
ZERO   DECB…………………ループ・カウンタ(Bレジスタ)を
                      デクリメント(−1)する
       BNE   LOOP……… 0でなければLOOPへ分岐する
       SWI ………………… プログラムを終了して，モニタへ
                      戻る
```

最初にＡレジスタに＄６Ｄ３０番地の内容をロードし，＄６Ｄ３１番地を０にクリアしておきます．また８回シフトしなければならないので，８をカウントするためにＢレジスタに８をロードしておきます．こうしておいてＡレジスタの内容を左にシフトすれば，ビットが１つＣフラグにはいりますから，これが０でないとき（つまり１のとき）＄６Ｄ３１番地をインクリメント（＋1）してやるのです．そして８回行ったかどうか知るためにＢレジスタを

デクリメント(−1)して，その内容が0であるか調べます．8回実行すればBレジスタは0になるので，そのときはZフラグが1になるはずです．そこでBNE命令，すなわちZフラグが0のときLOOPへ分岐して，いまの処理を繰り返します．

上のプログラムをハンドアセンブルしたものを次に示します．分岐するアドレスの計算に注意してください．

```
6D00   B6 6D 30          LDA   >$6D30
6D03   7F 6D 31          CLR   >$6D31
6D06   C6 08             LDB   #8
6D08   48          LOOP  LSLA
6D09   24 03             BCC   ZERO
6D0B   7C 6D 31          INC   >$6D31
6D0E   5A          ZERO  DECB
6D0F   26 F7             BNE   LOOP
6D11   3F                SWI
```

モニタでプログラムを入力し，実行結果を確認してみましょう．

Figure-13.4.2　実習14(条件分岐)の実行

```
*M6D00↵………プログラムを入力する
6D00 00-B6↵
6D01 00-6D↵
6D02 00-30↵
6D03 00-7F↵
6D04 00-6D↵
6D05 00-31↵
6D06 00-C6↵
6D07 00-08↵
6D08 00-48↵
6D09 00-24↵
6D0A 00-03↵
6D0B 00-7C↵
6D0C 00-6D↵
6D0D 00-31↵
6D0E 00-5A↵
6D0F 00-26↵
6D10 00-F7↵
6D11 00-3F↵
6D12 00-.↵

*M6D30↵………ビットの数を数えるデータ$F5を書き込む
6D30 00-F5↵
6D31 00-.↵

*D6D00↵………プログラム，データを確認する

6D00 B6 6D 30 7F 6D 31 C6 08  ――プログラム
6D08 48 24 03 7C 6D 31 5A 26
6D10 F7 3F 00 00 00 00 00 00
6D18 00 00 00 00 00 00 00 00
6D20 00 00 00 00 00 00 00 00
6D28 00 00 00 00 00 00 00 00
6D30 F5 00 00 00 00 00 00 00  ――"1"のビット数をカウントするデータ
6D38 00 00 00 00 00 00 00 00
*G6D00↵ ………プログラムを実行する

*D6D00↵………実行結果を確認する

6D00 B6 6D 30 7F 6D 31 C6 08
6D08 48 24 03 7C 6D 31 5A 26
6D10 F7 3F 00 00 00 00 00 00
6D18 00 00 00 00 00 00 00 00
6D20 00 00 00 00 00 00 00 00
6D28 00 00 00 00 00 00 00 00
6D30 F5 06 00 00 00 00 00 00  ――実行結果
6D38 00 00 00 00 00 00 00 00
*
```

$F5＝1111 0101
"1"が全部で6つあるので実行結果の$06は正しいことがわかる

## 実習15　条件判断とブランチ命令

CMP，TST 命令を使って，条件判断によるブランチ命令の実習を行います．次のプログラムは，＄６Ｅ３０番地の内容を＄６Ｅ３１番地の内容で割り，その余りを＄６Ｅ３２，商を＄６Ｅ３３番地へ格納するプログラムです．

```
6E00   B6  6E  30          LDA    >$6E30
6E03   5F                  CLRB
6E04   7D  6E  31          TST    >$6E31
6E07   27  0B              BEQ    ANS
6E09   B1  6E  31   LOOP   CMPA   >$6E31
6E0C   25  06              BCS    ANS
6E0E   B0  6E  31          SUBA   >$6E31
6E11   5C                  INCB
6E12   20  F5              BRA    LOOP
6E14   FD  6E  32   ANS    STD    >$6E32
6E17   3F                  SWI
```

このプログラムは，＄６Ｅ３０番地の内容をＡレジスタにロードしたら，引ける限り＄６Ｅ３１番地の内容を引き続け，そのたびにＢレジスタをインクリメント(＋1)してやります．こうすれば，最後には引いた回数つまり商がＢレジスタに求まり，余りがＡレジスタに残ります．また，＄６Ｅ０４～＄６Ｅ０８番地にはTST命令とBEQ命令がありますが，これは割る数が０の場合の処理(無限ループになるのを防ぐ)を行うためのものです．〝TST　>＄６Ｅ３１″は＄６Ｅ３１番地が０ならばＺフラグが１になります．

＄６Ｅ３０番地に被除数，＄６Ｅ３１番地に除数のデータをセットして，プログラムの実行結果を確認してください．

Figure-13.4.3 実習15(条件判断とブランチ命令)の実行

```
*M6E00↵………プログラムを入力する
6E00 00-B6↵
6E01 00-6E↵
6E02 00-30↵
6E03 00-5F↵
6E04 00-7D↵
6E05 00-6E↵
6E06 00-31↵
6E07 00-27↵
6E08 00-0B↵
6E09 00-B1↵
6E0A 00-6E↵
6E0B 00-31↵
  :
6E17 00-3F↵
6E18 00-.↵

*M6E30↵………被除数，除数のデータを入力する
6E30 00-F6↵
6E31 00-27↵
6E32 00-.↵

*D6E00↵………プログラム，データを確認する

6E00 B6 6E 30 5F 7D 6E 31 27
6E08 0B B1 6E 31 25 06 B0 6E ――プログラム
6E10 31 5C 20 F5 FD 6E 32 3F
6E18 00 00 00 00 00 00 00 00
6E20 00 00 00 00 00 00 00 00
6E28 00 00 00 00 00 00 00 00 ――被除数
6E30 F6 27 00 00 00 00 00 00 ――除数
6E38 00 00 00 00 00 00 00 00
*G6E00↵………プログラムを実行する

*D6E00↵………実行結果を確認する

6E00 B6 6E 30 5F 7D 6E 31 27
6E08 0B B1 6E 31 25 06 B0 6E
6E10 31 5C 20 F5 FD 6E 32 3F
6E18 00 00 00 00 00 00 00 00
6E20 00 00 00 00 00 00 00 00
6E28 00 00 00 00 00 00 00 00
6E30 F6 27 0C 06 00 00 00 00
6E38 00 00 00 00 00 00 00 00
*           |  └―商
            └――余り
```

# 14

# インデックスモードのアドレッシング方式

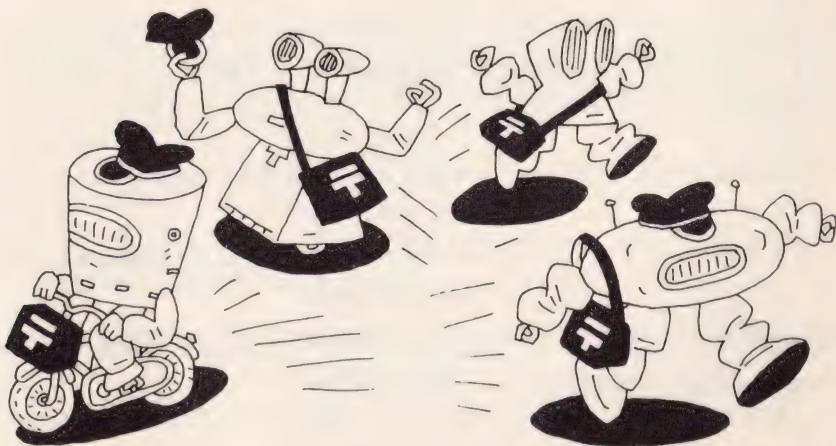
●アドレスを指定する場合，これまで解説してきたエクステンドやダイレクトといったモードでしかできないとすると，非常に不便を感じることがよくあります．そういう場合，ここで解説するインデックスモードを利用すると，アドレスの指定方法のバリエーションが一挙に拡大します．

6809が究極の8ビットCPUといわれるのもこのモードが存在するためで，インデックスモードを理解することが6809のマシン語を学ぶときの必須課題になります．

# 14 レジスタによるアドレス指定

前章までで6809の命令のほとんどは紹介してしまいましたが，最後にもう1つ，インデックスモードのアドレッシング方式という大切な内容が残っていますので，ここで説明しておきましょう．

6809を究極の8ビットCPUといわしめる最も大きな理由がこのインデックスモードのアドレッシングなのです．インデックスモードには6つのアドレッシング方式*1がありますが，本書では次に示す4つの方式を紹介します．

- ○定数オフセット
- ○アキュムレータ・オフセット
- ○オートインクリメント／オートデクリメント
- ○プログラム・カウンタ相対

*1 本書で紹介したもののほかに，インデックス・インダイレクト，エクステンド・インダイレクトがある

## “定数オフセット”

**オフセット**という言葉は，ある基準に対するずれのことで，ここではインデックス・レジスタの指すアドレスからのずれという意味で使われています．例えば，インデックス・レジスタに＄６０００がセットされているとすると，＄６０２０番地は＄６０００＋＄２０番地ですから，この＄２０が，基準となるアドレス（＄６０００）からのずれという意味でオフセットになります．

定数オフセットには，

① 0オフセット
② 5ビットオフセット
③ 8ビットオフセット
④ 16ビットオフセット

の4つがあり，〝ずれ〟の大きさに合せてこれらを使い分けることができます．いままで出てきた唯一のインデックス・アドレッシングである〝Xレジスタの指すアドレス〟は①の0オフセットのことだったのです．しかしながら6809のインデックスモードでは，Xレジスタだけではなく，Y，U，Sレジスタもまったく対等に使えるので，同様に〝Yレジスタの指すアドレス〟，〝Uレジスタの指すアドレス〟，〝Sレジスタの指すアドレス〟のような指定も行えます．これらをAレジスタへのLD命令を例にとって書いてみると次のようになります．

| | |
|---|---|
| LDA ,X | ……………Xレジスタの指すアドレスの内容をAレジスタにロードする |
| LDA ,Y | ……………Yレジスタの指すアドレスの内容をAレジスタにロードする |
| LDA ,U | ……………Uレジスタの指すアドレスの内容をAレジスタにロードする |
| LDA ,S | ……………Sレジスタの指すアドレスの内容をAレジスタにロードする |

さてこの例で"各レジスタの内容+1"のアドレスを指定したい場合，0オフセットではレジスタに1を足さなければなりませんが，②～④の定数オフセットを使えば，+1がオフセットということになり，次のように書くことができます．

LDA　　1，X……………X+1番地のアドレスの内容をAレジスタにロードする

こうするとXレジスタの内容を変化させることなく，X+1番地のアドレスのデータを参照することができるのです．また，5ビット，8ビット，16ビットと3種類もあるのは，オフセットの値が

| | | |
|---|---|---|
| −16～　+15 | のとき | 5ビットオフセット |
| −128～　+127 | のとき | 8ビットオフセット |
| −32768～+32767 | のとき | 16ビットオフセット |

のように使い分けるためで，次節で述べるように，できるだけビット数の少ないオフセットを使った方が命令に必要なバイト数が少なくて済み経済的です．

定数オフセットのグループ①～④の関係をFigure-14.1.1に示します．

Figure-14.1.1　定数オフセットによって参照できるアドレスの範囲

## “オートインクリメント／オートデクリメント”

オートインクリメント／オートデクリメントのモードは0オフセットの仲間ですが，インデックス・レジスタでアドレスを指定するのと同時にそのレジスタを+1, +2または−1, −2するものです．

例えば，$6000番地からの連続した100バイトを0にするような場合，エクステンドモードを使って書こうとすると，CLR命令を100個も書かなくてはならず現実的ではありませんが，インデックスモードのオートインクリメントを使えば，ループを作って100回繰り返してやればよいのです．

オートインクリメントを使ってこのプログラムを書くと，

```
       LDX    #$6000……$6000番地をXレジスタでポイン
                         トさせる
       LDB    #100…………ループの回数として100をBレジスタ
                         にロードする
       CLRA……………………Aレジスタをクリアする
LOOP   STA    ,X+…………Xレジスタの指すアドレスにAレジ
                         スタの内容($00)を入れた後，Xレ
                         ジスタに1を足す
       DECB……………………Bレジスタをデクリメント(−1)する
       BNE    LOOP…………Bレジスタの内容が0でなかったら
                         LOOPへ飛ぶ
```

のようになります．ループを回す前に，X，B，Aのレジスタには，それぞれクリアするメモリエリアのスタート・アドレス($6000)，クリアするバイト数(100)，0をセットしておけば，XレジスタはAレジスタをストアするべきアドレスを示した後，自動的に+1されて次のストアに備えることができます(Figure-14.1.2)．

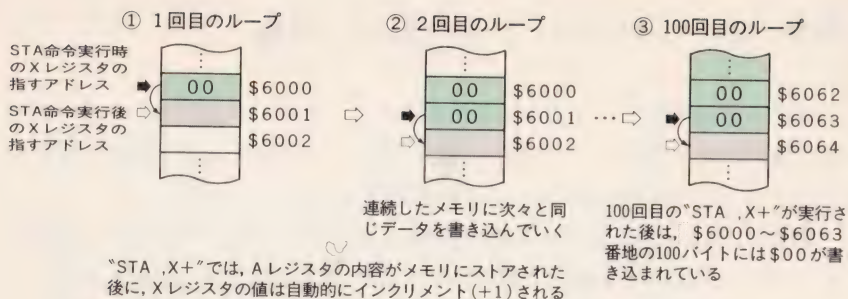



Figure-14.1.2　オートインクリメントを用いたプログラムの実行の様子

このプログラムでは，インデックス・レジスタの値が＋1されていましたが，このモードには，

① **オートインクリメント 1**
② **オートインクリメント 2**
③ **オートデクリメント 1**
④ **オートデクリメント 2**

の4種類があり，ここでは①のモードを使っています．

オートデクリメントはインデックス・レジスタを −1 するモードですが，1つだけ注意することがあります．それはオートインクリメントはレジスタが動作の対象となるアドレス(これを実効アドレス*1という)を示した後 +1 するのに対して，オートデクリメントは先に −1 してから実効アドレスを示すということです．これを**ポストインクリメント，プリデクリメント**といい，スタック・ポインタの動作の順序と同じになっています．

②，④はインデックス・レジスタをそれぞれ +2，−2 するモードで，

```
LDD     ,X++
STD     ,--X
```

のように16ビットのデータを扱うときに用いられます．

*1 インデックス・レジスタにオフセットを加えたアドレス

オートデクリメントはその動作の順序に合せて，アセンブリ言語で書くときも“，−−X”“，−X”という順番に書きます．Figure-14.1.3 にこの4つのモードの動作を表します．

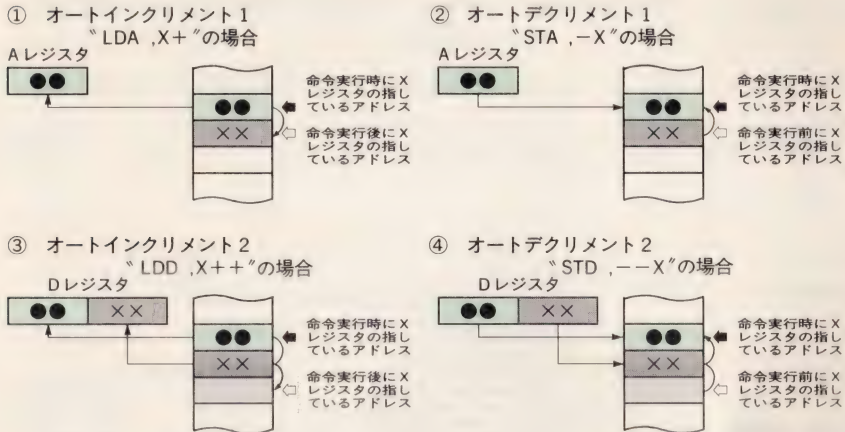



Figure-14.1.3 オートインクリメント/オートデクリメントの4つのモードの動作

## “アキュムレータ・オフセット”

インデックス・レジスタに付けるオフセットには，定数のほかにアキュムレータの内容で指定することもできます．アキュムレータ・オフセットには，

① Aオフセット
② Bオフセット
③ Dオフセット

の3種類があります．これらはアキュムレータの内容を2の補数表現で表されたデータとして，その値と各インデックス・レジスタとの和が実効アドレスになります．これらは，ニーモニックでは次のように表されます．

STB　A，X……………Bレジスタの内容をX+A番地にストアする
LDA　D，Y……………Y+D番地の内容をAレジスタにロードする
LDU　B，S……………S+B番地の内容をUレジスタにロードする

このモードは，配列などを扱うときに威力を発揮します．例えば＄６０００番地から１バイトずつのデータが並んでいる配列のｎ番目のデータを取り出すには，Ｂレジスタにｎを入れておいて，

ＬＤＸ　＃＄６０００……Ｘレジスタに配列の初めのアドレス値をロードする
ＬＤＡ　Ｂ，Ｘ……………Ｘ＋Ｂ番地の内容をＡレジスタにロードする

で簡単に実現できます．Figure-14.1.4 はアキュムレータ・オフセットの動作を表しています．

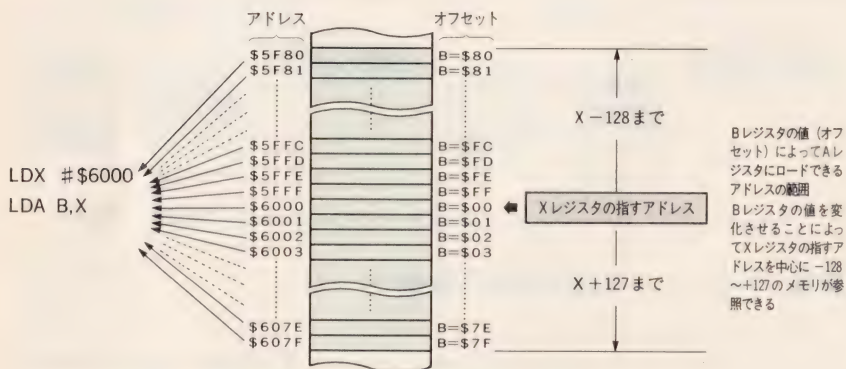



Figure-14.1.4　アキュムレータ・オフセットによるアドレス指定

## プログラム・カウンタ相対

このモードはプログラム・カウンタ(PC)をインデックス・レジスタとして扱い，実効アドレスを指定するものです．これには，

① プログラム・カウンタ相対８ビットオフセット
② プログラム・カウンタ相対16ビットオフセット

の2種類があり，プログラム・カウンタに8ビット，16ビットの定数オフセットを付けて表します．

プログラムをリロケータブルにするためには，分岐命令はすべてプログラム・カウンタからの距離(相対アドレス指定)で指定する必要があると述べましたが，これと同じことがデータのアドレス指定にもあてはまります．

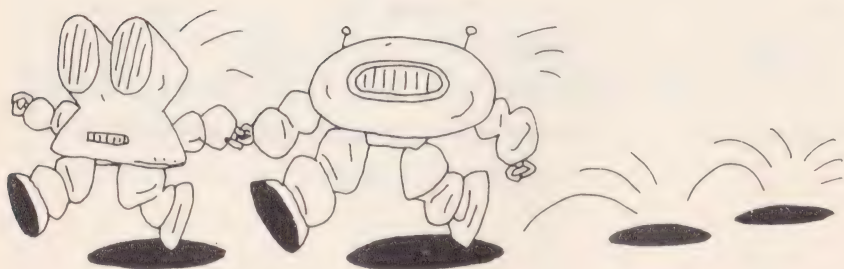
前章まで，実習のたびに何バイトかメモリ上にデータをセットするためのワークエリアをとり，それらを参照するときは常にエクステンドを使ってきました．しかし，この方法では，プログラムは移動できてもワークエリアは移動できません．そこでワークエリアも含めて完全にリロケータブルなプログラムを書くためには，ワークエリアを参照するためのアドレスも相対的にしなければなりません．つまり，ある命令がワークエリアのデータを参照する場合には，その命令が実行されるときのプログラム・カウンタの値からデータまでの距離(相対アドレス)をインデックスモードのプログラム・カウンタ相対によって指定するのです．

Figure-14.1.5は実習13のプログラムをリロケータブルになるように書き換えたものです．FDBという命令のようなものがありますが，これは6809に対する命令ではなくアセンブラに対しての命令で，そこに2バイトのワークエリアを確保して値をセットするという意味です．このプログラムはサブルーチンを呼ぶのにJSR命令ではなくBSR (Branch to SubRoutine)命令を使い，ワークエリアを参照するのにエクステンドではなくプログラム・カウンタ相対で行っています．リストの白い部分の"PCR"がプログラム・カウンタ相対を表しており，ソース・プログラムを書く段階では，図のようにして参照するデータのアドレスを示しておきます．ハンドアセンブルする場合には，ワークエリアのアドレスまでの距離(相対アドレス)で参照すべきアドレスを指定しますが，この距離は，ブランチ命令と同様に1バイト(または2バイト)の2の補数で表されます．なお，RND, SEED, WORK1, WORK2, WORK3とはサブルーチンやワークエリアに筆者が勝手に付けた名前です．

Figure-14.1.5 実習13をリロケータブルにしたプログラム

```
6C00    8D 11              BSR    RND          *1
6C02    ED 8D 0026         STD    WORK1,PCR
6C06    8D 0B              BSR    RND          ……PCR
6C08    ED 8D 0022         STD    WORK2,PCR    (Program Counter
6C0C    8D 05              BSR    RND           Relative)の略
6C0E    ED 8D 001E         STD    WORK3,PCR
6C12    3F                 SWI
                   *
6C13    EC 8D 0013  RND    LDD    SEED,PCR
6C17    84 08              ANDA   #$08
6C19    48                 ASLA
6C1A    48                 ASLA
6C1B    48                 ASLA
6C1C    48                 ASLA
6C1D    A8 8D 0009         EORA   SEED,PCR
6C21    58                 ASLB
6C22    49                 ROLA
6C23    C9 00              ADCB   #0
6C25    ED 8D 0001         STD    SEED,PCR
6C29    39                 RTS
                   *
6C2A       7053     SEED   FDB    $7053
6C2C       0000     WORK1  FDB    0
6C2E       0000     WORK2  FDB    0
6C30       0000     WORK3  FDB    0
```

＊1　この書式はアセンブラの書式であり，このように指定するとアセンブラでは自動的に（WORK1－PC）を計算して相対アドレスを出力する．ハンドアセンブルする場合には，"STD WORK1,PCR"が実行されたときのプログラム・カウンタの値（＄６Ｃ０６）からWORK1（＄６Ｃ２Ｃ） までの相対アドレスによってワークエリアを指定する．この場合は，＄２６（＄６Ｃ２Ａ－＄６Ｃ０６）が相対アドレスとなる．
命令実行後，Ｄレジスタの内容がWORK1のアドレス（＄６Ｃ２Ｃ，＄６Ｃ２Ｄ番地）にストアされる．

＊2　プログラム・カウンタ相対16ビットオフセットのポストバイト，次節および命令表８を参照．

# 14.2 ポストバイト

インデックス・アドレッシングにはいくつもの種類があるので，アドレッシングモードにインデックスを指定した命令(例えばLDAなら＄A 6)だけではどのモードなのかがわかりません．そこで＄A 6に続けてもう1バイト，すなわちポストバイトと呼ばれるコードが必要になります．以前，〝Xレジスタの指すアドレスをAレジスタにロードする〞という命令をアセンブルすると，

```
A6 84 ←――― LDA  ,X
   ︸
   ：
ポストバイト
```

のようになることは紹介しましたが，この場合の8 4がポストバイトです．命令表でLDA命令のインデックスモードを見ると〝A 6〞としか書いてありませんが，これと〝，X〞を意味する8 4とで1つの命令ができあがります．

巻末の命令表8でインデックスモードのポストバイトという表を見てください．この表が6809のインデックスモードの一覧表になるのですが，ポストバイトは，16進数ではなくビットパターンで書いてあります．そのためハンドアセンブルするときは自分で16進数に直さなければなりません．

では，ポストバイトの求め方を具体的に説明しましょう．まず〝，X〞の場合は0オフセットですから，表のその欄を見ると，

| 1 | R | R | 0 | 0 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|

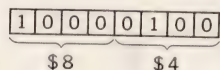
インデックス・レジスタ(X,Y,U,S)

RR
00＝X
01＝Y
10＝U
11＝S

と書いてあります．ここでRRというのは，4つあるインデックス・レジスタ

タ(X, Y, U, S)を区別するためのもので，表の横にあるようにXレジスタの場合は00をあてはめます．すると，

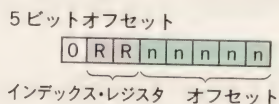
| 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|

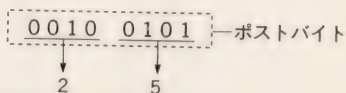
$8　$4

になり，これを16進数に直すと＄８４になります．

また，５ビットオフセットの場合，たとえば〝5，Y〞では，

5ビットオフセット

| 0 | R | R | n | n | n | n | n |
|---|---|---|---|---|---|---|---|

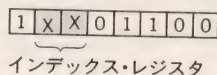
インデックス・レジスタ　オフセット

RRにYレジスタを示す01をあてはめ，nnnnnには2の補数表現で表したオフセットをあてはめます．この場合オフセットは5ですから，結局，

0010 0101 ―ポストバイト
2　5

になり，16進数では＄２５になります．

最後にもう１つ，プログラム・カウンタ相対8ビットオフセットのポストバイトについて説明しておきます．

巻末の表では，

| 1 | X | X | 0 | 1 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|

インデックス・レジスタ

となっていますが，XXのところは0,1どちらでもよいことを表しています．ですから00を入れればポストバイトは＄８Cになります．

さて，この例は8ビットオフセットですからポストバイトの次にオフセットを表すコードをもう1バイト置かなくてはなりませんが，このことはX，Y，U，Sなどの8ビットオフセットと変わりません．異なるのは，基準となるレジスタがプログラム・カウンタであり，プログラム・カウンタはプログラムの実行とともに常に変化している点です．つまり，同じアドレスを参

照するにもかかわらず，命令のあるアドレスによってこのオフセットの値が違ってくるのです．このことはブランチ(相対アドレス指定による分岐)命令と同様に考えれば納得できると思います．

ポストバイトを求めるのに初めは時間がかかると思いますが，慣れればそれほどでもありませんし，2進→16進変換のよい練習にもなります．Figure-14.2.1に定数オフセットとプログラム・カウンタ相対のポストバイトの求め方を図示しておきますので，他の命令についてもいろいろと試してみてください．

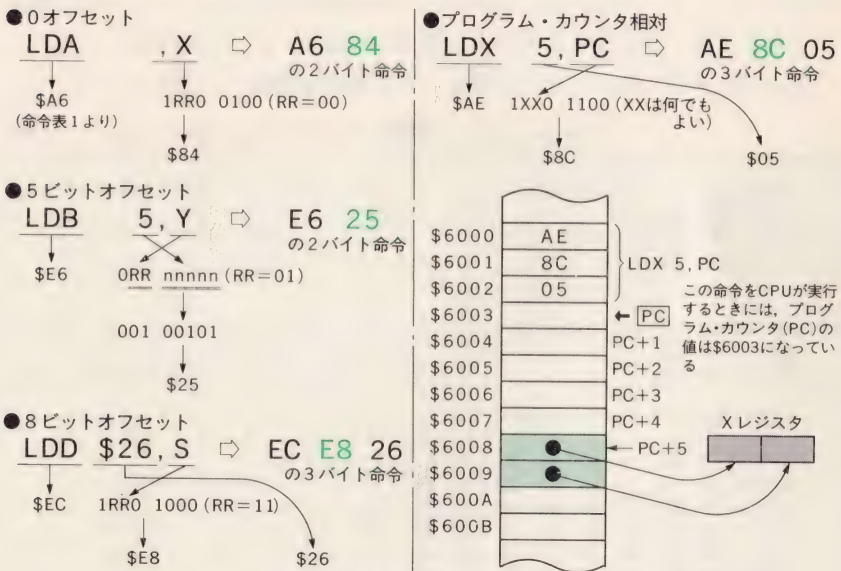



Figure-14.2.1 ポストバイト

## 実習16 オートインクリメントによるループ

インデックスモードでオートインクリメントを使ったプログラムの実習を行います．

次のプログラムは，＄７０００番地からの50バイトに＄ＡＡを書き込むものです．

```
        LDA   #$AA    ……Aレジスタに$AAをセットする
        LDB   #50     ……Bレジスタにループ回数をセットする
        LDX   #$7000  ……Xレジスタに$7000をセットする
→LOOP   STA   ,X+     ┐
        DECB          │ Aレジスタの内容を$7000番地
        BNE   LOOP    ┘ からの50バイトに書き込む
        SWI
```

Bレジスタで50を数えながら，Aレジスタの内容($AA)をXレジスタの指すアドレスへストアしています．このときオートインクリメントを指定してあるので，Xレジスタは1ずつ増えていきます．

上のプログラムをモニタで入力して実行結果を確認してみましょう．

Figure-14.2.1 実習16(オートインクリメントによるループ)の実行

```
*M6F00↵……プログラムを入力する
6F00 00-86↵
6F01 00-AA↵
6F02 00-C6↵
  :
6F0B 00-FB↵
6F0C 00-3F↵
6F0D 00-.↵

*D6F00↵……プログラムを確認する

6F00 86 AA C6 32 8E 70 00 A7 ――プログラム
6F08 80 5A 26 FB 3F 00 00 00
6F10 00 00 00 00 00 00 00 00
6F18 00 00 00 00 00 00 00 00
6F20 00 00 00 00 00 00 00 00
6F28 00 00 00 00 00 00 00 00
6F30 00 00 00 00 00 00 00 00
6F38 00 00 00 00 00 00 00 00
*G6F00↵……プログラムを実行する

*D7000↵……実行結果を確認する

7000 AA AA AA AA AA AA AA AA
7008 AA AA AA AA AA AA AA AA   Aレジスタにセットしたデータ($AA)が
7010 AA AA AA AA AA AA AA AA   $7000番地から50バイト
7018 AA AA AA AA AA AA AA AA   書き込まれている
7020 AA AA AA AA AA AA AA AA
7028 AA AA AA AA AA AA AA AA
7030 AA AA 00 00 00 00 00 00
7038 00 00 00 00 00 00 00 00
*
```

# 14・3 実効アドレスのロード LEA

6809のインデックスモードは，命令の対象となる実効アドレスを様々な方法で指定できますが，この実効アドレスそのものをレジスタにロードすることによってさらに応用範囲が広がります．

LEA(Load Effective Address)命令はそのための命令で，インデックス・レジスタX，Y，U，Sに実効アドレスをロードします．例えば，

```
LEAX  1, X
LEAY  B, U
LEAU  5, PC
LEAS  -4, S
```

とすれば，実効アドレスはそれぞれX＋1，U＋B，PC＋5，S－4であり，命令の意味は次のようになります．

```
LEAX  1, X     (X←X+1)
LEXY  B, U     (Y←U+B)
LEAU  5, PC    (U←PC+5)
LEAS  -4, S    (S←S-4)
```

つまり，本来アドレスを指定する目的で計算された実効アドレスをそのままレジスタにロードすれば，レジスタの加減算というまったく異なった目的に転用できるのです．

LEA命令の存在意義は大きく，特に〝LEAU　5，PC〞のような用法はリロケータブルなプログラムを書く上で非常に重要な意味を持っています．例えば，あるサブルーチンで配列を使用しなければならない場合，まずその先

頭アドレスをレジスタにロードしなければなりませんが，それを

```
LDU    #ARRAY
```

としてしまうと，ARRAYという固定的な値(アドレス)をUレジスタにロードすることになり，このプログラムおよびワークエリア(配列)はリロケータブルになりません．そこで

```
LEAU  ARRAY, PCR
```

としてやれば，マシン語では命令のあるアドレスからARRAYまでの距離がプログラム・カウンタに対するオフセットとして与えられるので，プログラム(およびワークエリア)の置かれるアドレスによらず，Uレジスタには正しくARRAYの先頭アドレスがロードされます．

＄6000番地から始まるプログラムを＄7000番地に移動した場合を例に，この違いをFigure-14.3.1に示しておきます．

① LDU　＃ARRAYの場合
(ARRAYは配列の先頭アドレス)

$6000 CE
$6001 60
$6002 10
$6003
⋮
$6010 ← ARRAY
$6011

LDU　＃ARRAY
(LDU　＃6010)
このとき
U＝$6010

プログラムを
＄7000番
地へ移動する

$7000 CE
$7001 60
$7002 10
$7003
⋮
$7010
$7011

LDU　＃$6010
イミディエイトのデータは本来＄7010でなければならないが，LD命令では実効アドレスを得ることができない

② LEAU ARRAY, PCRの場合

$6000 33
$6001 8C
$6002 0D
$6003
⋮
$6010 ←ARRAY
$6011

LEAU ARRAY, PCR
(LEAU $0D, PC)
このとき
U＝$6010
(PC＋$0D)

＄6003番地から＄6010番地までの距離＄0D（＝13）

プログラムを
＄7000番
地に移動する

$7000 33
$7001 8C
$7002 0D
$7003
⋮
$7010 ←ARRAY
$7011

LEAU $0D, PC
このとき
U＝$7010
(PC＋$0D)

プログラムの置かれるアドレスが移動しても，LEA命令を使っていれば相対的なアドレス指定ができる

Figure-14.3.1　プログラム・カウンタ相対を使ってリロケータブルにする方法

# 15 やさしいプログラム例

●前章までで命令の解説は終わり，いよいよ実際にプログラムを作ってみることにします．これから皆さんがプログラムを組むにあたって，参考になるようなものを選んで掲載しました．

本章で紹介するマシン語のプログラムは入出力ルーチン，除算ルーチン，ソーティングルーチンの3つです．キーボードからの入力やスクリーンへの出力は機種によって異なりますが，各機種別に紹介してありますので，どのマシンでも使うことができます．これらのルーチンは今後みなさんのアプリケーション・プログラムで大いに利用してください．

最後のサンプル・プログラムはBASICプログラムとマシン語プログラムを結合したものです．BASICとマシン語の結合はなかなか面倒なものなので，パラメータの受け渡しやマシン語エリアの確保などに注意してください．

なお，本章のリストはすべてアセンブラによって出力したものを掲載しています．

# 15 入出力ルーチン

マシン語でプログラムを組むとき，最初に問題になるのが入出力の部分だと思います．そこで最も基本的な，

○ **キーボードからの1文字入力**　(GETCH)
○ **スクリーンへの1文字出力**　(PUTCH)

の2つを扱ったプログラムを **Figure-15.1.1** に紹介します．

Figure-15.1.1　キーボードから入力された文字を10回表示するプログラム

このプログラムはキーボードから１文字を入力するとその文字を10個スクリーンに出力し，エスケープコード（＄１B）が入力されるとモニタに戻るようになっています．実際に入力／出力している部分は〝JSR　>GETCH〞，〝JSR　>PUTCH〞で，これらはROM内ルーチン*1を呼んでいます．GETCHのルーチンは，入力された文字のアスキーコードがAレジスタにはいって戻ってくるもので，PUTCHのルーチンはAレジスタの内容をアスキーコードとみなしてその文字をスクリーンに出力するものです．

GETCH，PUTCHはこれらの処理をするルーチンのエントリ・アドレスを表していますが，機種によって異なります．Figure-15.1.1はFM-7用のアセンブル結果ですが，単にGETCH，PUTCHの値が違うだけなので，他の機種のエントリ・アドレスをFigure-15.1.2にまとめておきます．

| 機種 ／ ROM内ルーチン | LEVEL 3 (S1のBモード含む) | FM-8 | FM-7/77/NEW7 |
|---|---|---|---|
| GETCH | E804 | D336 | D072 |
| PUTCH | E820 | D352 | D08E |

Figure-15.1.2　GETCH／PUTCH エントリ・アドレス

Figure-15.1.1のダンプリストと実行結果を，Figure-15.1.3に示します．

Figure-15.1.3　プログラムの実行例

```
*D6000⏎………プログラムを表示する

6000 86 2A BD D0 8E BD D0 72
6008 81 1B 27 0A C6 0A BD D0 ←プログラム
6010 8E 5A 26 FA 20 EA 3F 00
6018 00 00 00 00 00 00 00 00
6020 00 00 00 00 00 00 00 00
6028 00 00 00 00 00 00 00 00
6030 00 00 00 00 00 00 00 00
6038 00 00 00 00 00 00 00 00
*G6000⏎………$6000番地からプログラムを実行する
*++++++++++*==========*,,,,,,,,,,*------
----*!!!!!!!!!!*~~~~~~~~~~*¥¥¥¥¥¥¥¥¥¥*$$
$$$$$$$$*%%%%%%%%%%*((((((((((*))))))))))
)*FFFFFFFFFF*NNNNNNNNNN*VVVVVVVVVV*KKKKK
KKKKK*EEEEEEEEEE*;;;;;;;;;;* ESC  プロンプトの"*"を表示して入力を待っている
*                 └入力した文字を繰り返して10文字表示する
└ ESC キーを押すとモニタに戻る．このプロンプトはモニタのプロンプトである
```

*1　BASICインタープリタに組み込まれているマシン語のサブルーチン

# 15.2 除算ルーチン

6809には除算命令というものがありませんので，マシン語で除算をするには自分でプログラムを組む必要があります．しかし，除算をするプログラムは案外面倒なので，ここに簡単な除算ルーチンの例を紹介しておきましょう．

Figure-15.2.1は除算をするプログラムですが，これは単なるサブルーチンですから，皆さんの作るメインルーチンから呼ぶことによって，初めて除算をさせることができます．つまり除数と被除数をこのサブルーチンに渡してやらなければなりません*1．

このサブルーチンは16ビットの値を8ビットで割って，商と余りをそれぞれ8ビットで得るもので，値はすべて符号なしとして扱っています．具体的には被除数をXレジスタに，除数をBレジスタに持ってこのサブルーチンを呼びます．すると商をAレジスタに，余りをBレジスタに持って戻ってきますから，後は自分の好きなようにそれを利用すればよいのです．なお，0で割ったり，商が8ビットを超えるような除算はできませんが，このようなときはCフラグが0になって戻ってきますから，BCC命令でそれを調べることによって正常に計算できたかどうかを判断できます．正常なときはCフラグが1になっています．

このプログラムの動作はなかなか複雑で，理解に時間がかかると思いますが，自分がCPUになったつもりで1つ1つ命令を追っていけば必ずわかるはずです．

特にスタック上にとったワークエリアで，商を負論理で計算している点に注意してください．負論理で得た商は，最後にCOM命令で正論理に戻していますが，このときCフラグが常に1にセットされることを利用して，メインルーチンへ正常終了の情報として渡しているのです．

*1 このようにメインルーチンとサブルーチンの間でやり取りするデータのことを，引数あるいはパラメータと呼ぶ

Figure-15.2.1　除算ルーチンのアセンブルリスト

```
                         *
                         *          div
                         *
      (6000)                   ORG    $6000       Cフラグが1のときは正常終了
                         *
6000   4F                 DIV  CLRA               除数の上位8ビットをクリアする
6001   34 16                   PSHS   A,B,X       被除数と除数をプッシュする
6003   C6 08                   LDB    #8          ループ・カウンタとして8をセットする
6005   34 04                   PSHS   B           ループ・カウンタをプッシュする
6007   E6 63                   LDB    3,S         被除数の上位8ビットをBレジスタにロードする
6009   E1 62                   CMPB   2,S         被除数の上位8ビットが除数を超えた場合はオーバーフローなので
600B   24 13                   BCC    DIVEND      DIVENDへ分岐する
                         *
600D   68 64             LOOP  LSL    4,S
600F   59                      ROLB               被除数を左へシフトする
6010   49                      ROLA
6011   A3 61                   SUBD   1,S         被除数の上位8ビットを除数と比較する
6013   24 02                   BCC    BIG         被除数≧除数ならばBIGへ分岐する
6015   E3 61                   ADDD   1,S         被除数<除数ならば引いた数を加える このときC=1となる
6017   69 63             BIG   ROL    3,S         商のビットをセットする．商は負論理
6019   6A E4                   DEC    ,S          (引けたときはC=0，引けないときはC=1)で計算する
601B   26 F0                   BNE    LOOP        ループ・カウンタが0にならない間はLOOPへ分岐する
                         *
601D   A6 63                   LDA    3,S         商をAレジスタにロードする
601F   43                      COMA               Aを正論理にするとともにC=1にする
                         *
6020   32 65             DIVEND LEAS  5,S         ……ローカルワークエリアを消去する
6022   39                      RTS                サブルーチンから戻る
                         *
      (6022)                   END
```

Figure-15.2.2　除算ルーチンのダンプリスト

```
*D6000↵     $6000番地からプログラムを表示する

6000 4F 34 16 C6 08 34 04 E6
6008 63 E1 62 24 13 68 64 59     除算サブルーチンのプログラム
6010 49 A3 61 24 02 E3 61 69
6018 63 6A E4 26 F0 A6 63 43
6020 32 65 39 00 00 00 00 00
6028 00 00 00 00 00 00 00 00
6030 00 00 00 00 00 00 00 00
6038 00 00 00 00 00 00 00 00
*
```

# 15.3 データの並べ換え

でたらめに並んでいるデータをある規則(大きい順，小さい順，アルファベット順など)に従って並べ替えることを**ソーティング**といいます．大量のデータをソーティングしておけば，必要なデータを素早く検索できるため，データ処理をスムースに行うことができます．もちろんBASICでもソーティング・プログラムを作ることはできますが，10バイトや20バイトのデータならばいざしらず，1000バイト，2000バイトのデータをソーティングするには分単位で時間がかかるため実用的ではありません．こんなときこそ，マシン語を利用すれば，その〝高速性〟がフルに発揮できます．

ここで紹介するのは，符号付きの2バイトのデータの配列を小さい順にソーティングをするプログラムです．このプログラムもサブルーチンになっているので単独では利用できませんが，リロケータブルですのでどんなプログラムにもそのまま組み込んで使うことができます．

このサブルーチンを利用するには，

① データ配列(メモリに置かれたもの)
② 配列の先頭アドレス
③ データの個数

の3つをメインルーチンで用意する必要があります．サブルーチンにパラメータを受け渡す場合には，Figure-15.3.1に示すように，②と③の値をそれぞれUレジスタ，Dレジスタに入れて，サブルーチンを呼ばなくてはなりません．Figure-15.3.2，15.3.3がソート・サブルーチンのアセンブルリストとダンプリストです．また，このマシン語プログラムと同様の働きをするBASICのサブルーチンをFigure-15.3.4に示しておきましたので，アルゴリズムや変数の使われ方を調べる際の参考にしてください．

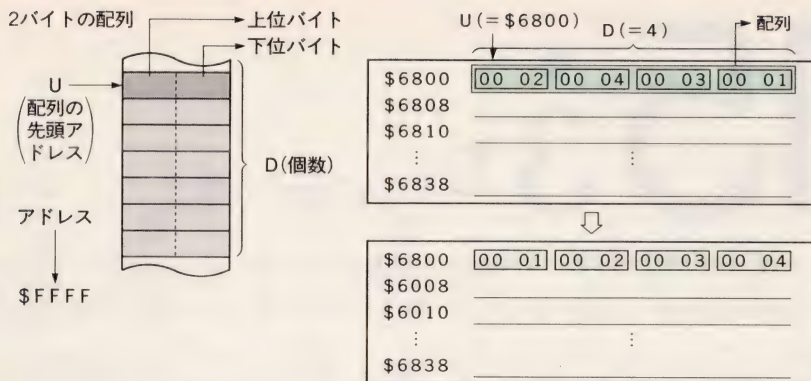



Figure-15.3.1　配列とレジスタの設定の仕方

Figure-15.3.2　ソートルーチンのアセンブルリスト

```
                        *
                        *          sort
                        *
         (6000)                 ORG    $6000
                        *
6000     34 76           SORT   PSHS   D,X,Y,U      レジスタの退避
6002     58                     LSLB
6003     49                     ROLA                を代入する
6004     31 CB                  LEAY   D,U
6006     20 17                  BRA    START
                        *
6008     30 A4           LOOP2  LEAX   ,Y
600A     33 A4                  LEAU   ,Y
600C   10A3 83           LOOP1  CMPD   ,--X
600F     2C 04                  BGE    BIG
6011     EC 84                  LDD    ,X
6013     33 84                  LEAU   ,X
6015     AC 66           BIG    CMPX   6,S          先頭のデータを比較するまで
6017     26 F3                  BNE    LOOP1        LOOP1に分岐する
6019     AE A4                  LDX    ,Y           最大値をYレジスタの指すアド
601B     ED A4                  STD    ,Y           レスの内容と交換する
601D     AF C4                  STX    ,U
601F     EC A3           START  LDD    ,--Y
6021   10AC 66                  CMPY   6,S          先頭のデータになるまで繰り返す
6024     22 E2                  BHI    LOOP2
6026     35 F6                  PULS   D,X,Y,U,PC   レジスタを復帰して
                        *                           このサブルーチンか
         (6027)                 END                 ら返る
```

Figure-15.3.3 ソートルーチンのダンプリスト

```
*D6000↲ ……… $6000番地から表示する

6000 34 76 58 49 31 CB 20 17
6008 30 A4 33 A4 10 A3 83 2C
6010 04 EC 84 33 84 AC 66 26 ── ソートのマシン語サブルーチン
6018 F3 AE A4 ED A4 AF C4 EC
6020 A3 10 AC 66 26 E2 35 F6
6028 00 00 00 00 00 00 00 00
6030 00 00 00 00 00 00 00 00
6038 00 00 00 00 00 00 00 00
*
```

Figure-15.3.4 ソートルーチンと同様の働きをするBASICサブルーチン

```
1000 START=1
1010 ITEM=100 ……… データの数を100個とする(これはメインルーチンで設定する)
1020 '
1030 Y=START+ITEM ……… (最大の添字＋1)をYに代入する
1040 GOTO 1170 :'START
1050 'LOOP2
1060   X=Y   } ポインタの設定
1070   U=Y   }
1080 'LOOP1
1090   X=X-1 : IF D>=ARRAY(X) THEN 1120 :'BIG
1100   D=ARRAY(X) } DよりARRAY(X)の方が大きければ，その値とポインタを
1110   U=X        } 新たな最大値とする
1120 'BIG
1130   IF X<>START THEN 1080 :'LOOP1
1140   X=ARRAY(Y)   }
1150   ARRAY(Y)=D   } 最大のデータとARRAY(Y)を交換する
1160   ARRAY(U)=X   }
1170 'START
1180   Y=Y-1 : D=ARRAY(Y) ……… 次の最大値をARRAY(Y)とする
1190   IF Y>START THEN 1050 :'LOOP2 ……… 比べるデータが残っているなら
1200  RETURN ……… メインルーチンへ戻る          1050行に分岐する
```

# 15・4 BASICとマシン語の結合

マシン語プログラムは高速な処理ができるといった利点がある反面，ちょっとした処理をするにも細かなことまでプログラムしなければならず，その開発に時間がかかるという欠点があります．そこで，高速な処理が要求されるような部分をマシン語で行い，それ以外の部分をBASICで行うようにすれば，これらの欠点を相互に補うことができます．

そこで本節では，BASICプログラムとマシン語プログラムを共存させて利用する方法について説明していきます．

## “BASICとマシン語プログラムの関係”

BASICプログラムとマシン語プログラムを1つのプログラムとして作る場合，常にBASICプログラムがメインルーチンとなり，マシン語のプログラムはサブルーチンとして呼び出されます．しかし，もともとBASICとマシン語とではコンピュータにおけるプログラムの形態が異なるため，BASICプログラムとマシン語プログラムを結合するには，いくつか知識が必要です．それは，

① マシン語のプログラムを置くメモリを，BASICから干渉を受けないように確保する
② マシン語のサブルーチンをBASICプログラムから呼べるようにする
③ BASICプログラムとマシン語の間で，データの受け渡しを行う

の3点です．

## マシン語プログラムの保護

電源 ON で BASIC インタープリタが起動した時点では，すべてのメモリ(64 K バイト)は BASIC インタープリタによって管理されています．ですから BASIC インタープリタが BASIC プログラムを実行するために必要なワークエリア，変数エリアなどにメモリを使用するために，単にマシン語プログラムをメモリにおくだけでは，いつ BASIC インタープリタにそのエリアを侵されるとも限りません．そのため BASIC のプログラムとマシン語のプログラムをいっしょに使う場合には，BASIC インタープリタによってマシン語が破壊されないように，マシン語のためのメモリエリアを確保する必要があります．

マシン語のプログラムエリアを確保するためには，BASIC の CLEAR 文を使います．これは，

```
CLEAR 300,&H5FFF
```

とすると，Figure-15.4.1 に示すように＄６０００番地から RAM エリアの終わり(FM-7 の場合，ROM バージョンならば＄７ＦＦＦ)までがマシン語のためのエリアとして確保されたことになります(巻末の APPENDIX 参照)．

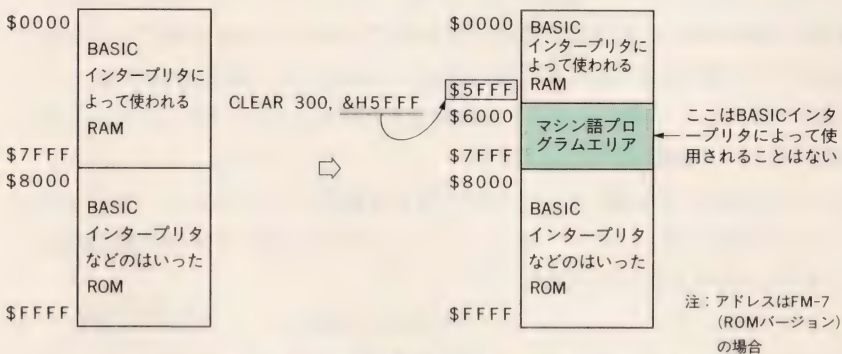



Figure-15.4.1 CLEAR命令によるマシン語エリアの確保

## マシン語プログラムの呼び出し

BASICからマシン語サブルーチンを呼び出すには，

① EXEC命令による方法
② USR関数による方法

の2通りがあります．

EXEC命令は単にEXECに続けてマシン語のプログラムのスタート・アドレスを指定すればよいので，

```
EXEC  &H6000
```

とすれば＄6000番地から始まるマシン語プログラムをサブルーチンとして呼ぶことができます．

USR関数を用いる場合は，いつもDEFUSR文とともに使われます．DEFUSR文はマシン語プログラムのスタート・アドレスを指定する命令で，マシン語プログラムが＄6000番地から始まるのであれば，次のように書きます．

```
DEFUSR=&H6000
N=USR(M)
```

USR関数の後ろのカッコのなかに書かれているのは，マシン語プログラムに受け渡す引数*1です．上の例では，BASICプログラムで使っているMという変数の値がマシン語のプログラムに渡され，マシン語のプログラムから得た値をNという変数に代入します．

このように，BASICとマシン語で値を受け取ったり返したりする場合にはUSR関数を使い，マシン語のルーチンを単に起動するだけの場合などにはEXEC命令を使うのが便利です．

マシン語プログラム(サブルーチン)からBASICプログラムへ戻るときは，マシン語から呼ばれたときと同様，RTS命令で戻ります(Figure-15.4.2)．

*1 BASICのメインルーチンからマシン語サブルーチンに受け渡すデータ，この場合は変数Mの値

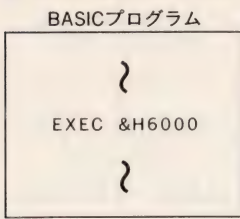
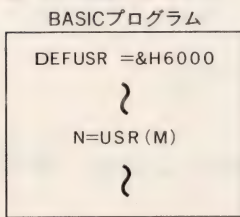



Figure-15.4.2　マシン語サブルーチンの呼び出し

## “データの受け渡し”

BASIC で扱えるデータの型は，整数，単精度実数，倍精度実数，文字列の4種類があり，どの型のデータでもマシン語プログラムと受け渡しができます．しかし実数型データの処理は複雑なので，ここでは整数型のデータと文字列の受け渡しについてのみ解説します．

BASIC で扱う整数型のデータとは，符号付き2バイトのデータのことであり，マシン語でも容易に処理することが可能です．M，N を整数型の変数に宣言しておいて，

N＝USR(M)

とすると，Figure-15.4.2に示したように“Xレジスタの内容＋2”番地(およびその次のアドレス)にMの値がはいり，マシン語プログラムに実行が移されます．マシン語プログラムではその値を，

```
LDD    2，X……………X+2番地の内容をDレジスタにロードする
```

のようにして得ることができ，マシン語プログラムからBASICに値を返したければ，

```
STD    2，X……………Dレジスタの内容をX+2番地へストアする
```

とすれば，Dレジスタの内容をBASICに渡すことができます．
また，文字列の場合には，

```
B$=USR(A$)
```

とすると，Xレジスタは**ストリング・ディスクリプタ**と呼ばれる3バイトのデータのスタート・アドレスを指します．すなわち，Xレジスタの指すアドレスに文字列の長さ，“Xレジスタの内容＋1”のアドレス(およびその次のアドレス)に文字列の格納されている先頭アドレスがはいり，マシン語プログラムが実行されます．マシン語プログラムでは文字列の長さは，

```
LDB     ，X……………X番地の内容をBレジスタにロードする
```

で，また，文字列の格納されている先頭アドレスは

```
LDU    1，X…………X＋1番地の内容をUレジスタにロードする
```

で得ることができます．これらの様子をFigure-15.4.3に示します．

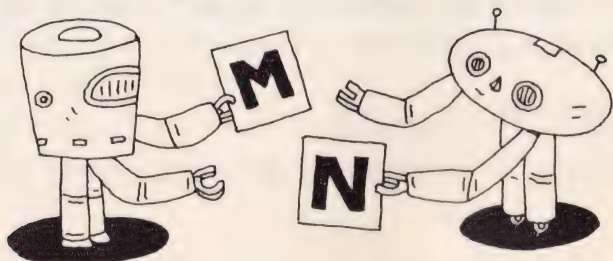
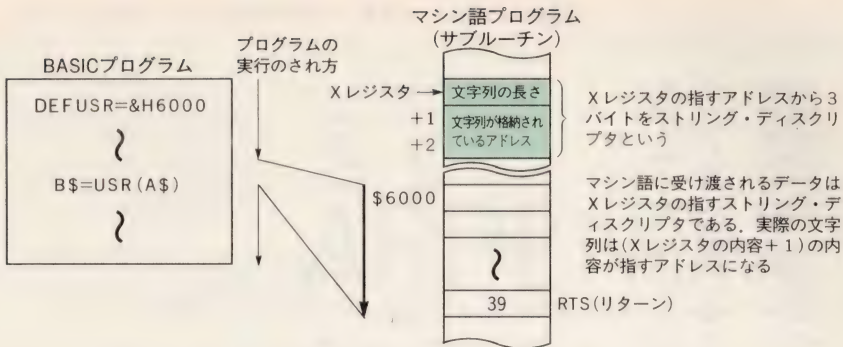



Figure-15.4.3　引数が文字列のときのマシン語ルーチンの呼び出し

## BASICとの結合の実際

ここではBASICからマシン語を利用する例として，文字列中に含まれている英小文字を大文字に変換するマシン語のサブルーチンをBASICから呼び出せるようにしてみましょう．Figure-15.4.4がメインルーチンとなるBASICのプログラムですが，BASICの部分では文字列の入力を受けて，それを引数としてFigure-15.4.5のマシン語のサブルーチンに受け渡します．このマシン語のサブルーチンは，さらに下位のサブルーチンとしてAレジスタの1文字だけ英小文字を大文字に変換するサブルーチンを呼び出す構造になっているので，受け取った引数を変換した後にBASICプログラムへ戻し，スクリーンに表示しています．

Figure-15.4.6がマシン語サブルーチンのダンプリストですので，これをモニタで入力した後にBASICプログラムを実行してください．

Figure-15.4.4　BASICのメインルーチン

```
100 CLEAR 600,&H5FFF ………文字列領域の大きさを600バイト，BASICの使用する上限のメモリアドレスを＄５ＦＦＦ番地に設定する
110 DEF USR=&H6000 …………マシン語ルーチンのスタート・アドレスをＵＳＲ０に設定する
120 LINE INPUT A$ ……………Ａ＄に１行入力する
130   IF A$="" THEN END ………RETURN のみ入力するとプログラムを終了する
140   A$=USR(A$) ………小文字を大文字に変換するマシン語ルーチンを呼び出す
150   PRINT A$ …………変換した文字列を表示する
160 GOTO 120 ……………次の行の入力に移る
```

Figure-15.4.5　小文字→大文字変換を行うマシン語サブルーチン

```
6000    34 76      PROG    PSHS  D,X,Y,U…レジスタの退避
6002    81 03              CMPA  #$03      引数の型が文字型以外なら何も
6004    26 0F              BNE   ERROR     せずBASICに戻る
6006    E6 84              LDB   ,X        …文字列の長さをBレジスタにロードする
6008    27 0B              BEQ   ERROR     …引数が""(ヌルストリング)なら
                                             何もせずBASICに戻る
600A    EE 01              LDU   1,X       …文字列の先頭アドレスをUレジ
                                             スタにロードする
*
600C    A6 C4      LOOP    LDA   ,U        …Aレジスタに1文字ロードする
600E    8D 07              BSR   UPPER…小文字を大文字にするサブルー
                                         チンを呼ぶ
6010    A7 C0              STA   ,U+       …変換した結果をストアしてポイ
                                             ンタを1つ進める
6012    5A                 DECB          ⎫
6013    26 F7              BNE   LOOP    ⎭ 文字数
6015    35 F6      ERROR   PULS  D,X,Y,U,PC…レジスタを復改して
                                             BASICに戻る
*
6017    81 61      UPPER   CMPA  #$61 …"a"と比較する
6019    25 06              BCS   NOTLOW
601B    81 7B              CMPA  #$7B …"z"+1と比較する
601D    24 02              BCC   NOTLOW
601F    80 20              SUBA  #$20 …小文字なら$20を引いて大文
                                        字に変換する
6021    39         NOTLOW  RTS …サブルーチンから戻る
```

Figure-15.4.6　マシン語サブルーチンのダンプリストと実行結果

```
*D6000↲………$6000番地からマシン語プログラムを表示する

6000 34 76 81 03 26 0F E6 84
6008 27 0B EE 01 A6 C4 8D 07
6010 A7 C0 5A 26 F7 35 F6 81   ──マシン語プログラム
6018 61 25 06 81 7B 24 02 80
6020 20 39 00 00 00 00 00 00
6028 00 00 00 00 00 00 00 00
6030 00 00 00 00 00 00 00 00
6038 00 00 00 00 00 00 00 00
*BREAK ………BREAKキーでモニタからBASICのコマンドレベルに戻る

Break
Ready
run↲………BASICプログラムを起動する
This is the sample program for "upper".↲     入力した文字列のうち小文字
THIS IS THE SAMPLE PROGRAM FOR "UPPER".      の部分が大文字に変換される
Small characters are replaced with CAPIT     記号や大文字は変わらない
AL through upper filter.↲
SMALL CHARACTERS ARE REPLACED WITH CAPIT
AL THROUGH UPPER FILTER.
↲……リターンキーのみ入力するとプログラムを終了する
Ready
```

# APPENDIX

## 1．SWI命令と初期設定について

本書では，実習等でプログラムを入力する際，常にその終わりにSWI命令（＄３Ｆ）を付けており，これを終わりを意味する命令，もしくはモニタに戻る命令と説明していましたが，ここでその種明かしをしておきましょう．

SWI（SoftWare Interupt（ソフトウェア インタラプト））命令とは，ソフトウェアによって割込みを起こさせる命令なのですが，割込みに関しては本書の知識だけでは解説しきれないことなので，ここではFM-7を例にSWI命令の動作だけを説明することにします．

まず，Figure-A.1.1を見てください．この図はCPUがSWI命令を読み込んだときの動作を表しています．

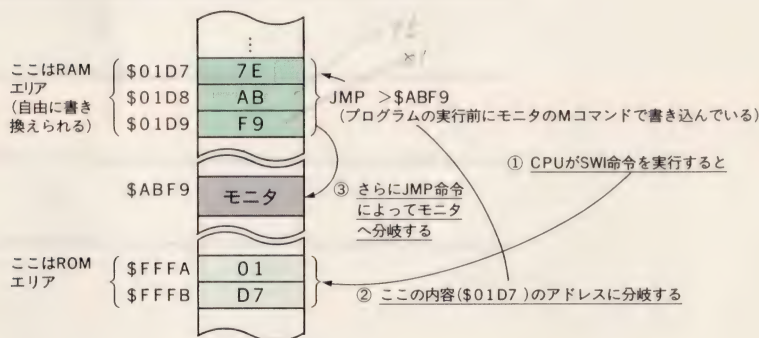



Figure- A.1.1 SWI命令(FM-7の例)

Ｓレジスタを除くすべてのレジスタをシステムスタックへプッシュ（PSHS CC，Ａ，Ｂ，DP，Ｘ，Ｙ，Ｕ，PCと同じ）した後，＄ＦＦＦＡ，＄ＦＦＦＢ番地に書かれているアドレスへ分岐するのです．この分岐先のアドレスは，機種によって異なりますが，FM-7の場合には＄０１，＄Ｄ７が書き込まれており，みなさんの入力したプログラムがSWI命令まで実行されると，CPUは＄０１Ｄ７番地へ分岐します．

試しにみなさんのコンピュータの＄ＦＦＦＡ，＄ＦＦＦＢ番地の内容をＤコマンドで確認してください．そこには＄０１，＄Ｄ７(FM-7の場合)と書かれているはずです．

さて，＄０１Ｄ７番地に分岐した後はいったいどうなるのでしょうか．ここで，１章で行った初期設定をもう一度思い出してください．初期設定では，＄０１Ｄ７番地(FM-7の場合)に＄７Ｅ，＄ＡＢ，＄Ｆ９(この３バイトはニーモニックで書くと，JMP ＞＄ＡＢＦ９)と書き込みましたが，＄０１Ｄ７番地に分岐した後はこの命令が実行されます．つまり，＄ＡＢＦ９番地に分岐するわけですが，このアドレスはモニタ・プログラムのエントリ・アドレス(FM-7の場合)であるため，SWI命令を実行した後は，いつもモニタのコマンド待ちの状態に戻るのです．

次に，SWI命令がなぜ＄ＦＦＦＡ，＄ＦＦＦＢ番地などという半端なアドレスを参照するかについて，少しだけ触れておきましょう．

6809には**割込みベクタ**(１つのアドレスを指し示す２バイトのデータ)と呼ばれるものが７つ存在しますが，これらのベクタは＄ＦＦＦ２〜＄ＦＦＦＦ番地に書いておくことになっています．SWI命令が実行されたときに参照するベクタは，いつも＄ＦＦＦＡ，＄ＦＦＦＢ番地の内容に決まっているのです．

SWI命令以外で割込みベクタを使うものの１つに，RESET(リセット)があります．RESETとは，コンピュータのリセットボタンを押したときや電源を入れたときなどにCPUが行うもので，CPUの状態を初期化することです．RESET後は，すぐプログラムを実行し始めるのですが，その際にCPUに何番地からのプログラムを実行するのか教えなければなりません．

6809ではこのアドレス(ベクタ)を＄ＦＦＦＥ，＄ＦＦＦＦ番地に書いておくことになっており，CPUはRESETされるとまず初めに＄ＦＦＦＥ，＄ＦＦＦＦ番地を読んで，そこに書かれているアドレスをプログラム・カウンタにセットすることによってプログラムの実行が開始されます．FM-7の場合はここに〝＄ＦＥ００〟と書かれており，それによってRESET直後にCPUが実行するプログラムのスタート・アドレスを知ることができます．

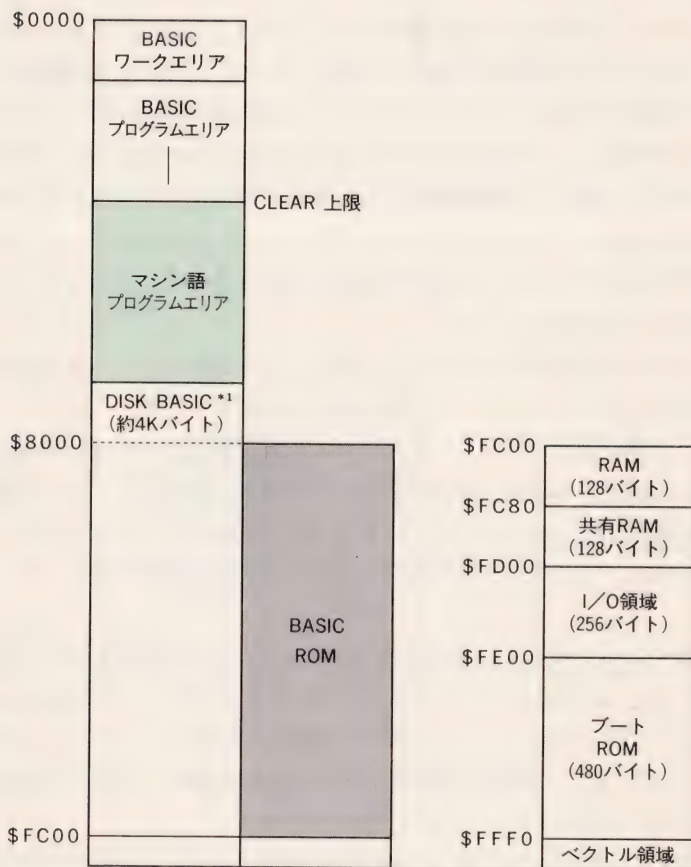
## 2. 機種別メモリマップ

＊1 DISK BASICを使用しない場合はプログラム領域として使用できる

Figure-A.2.1 FM-8,FM-7/77/NEW7メモリマップ

＊1　グラフィックRAMの大きさは起動時の設定およびNEWONによって変化する

＊2　S1のBモード,もしくはLEVEL-3でRAMボードをつけた場合はプログラム領域として使用できる

＊3　DISK BASICを使用しない場合はプログラム領域として使用できる

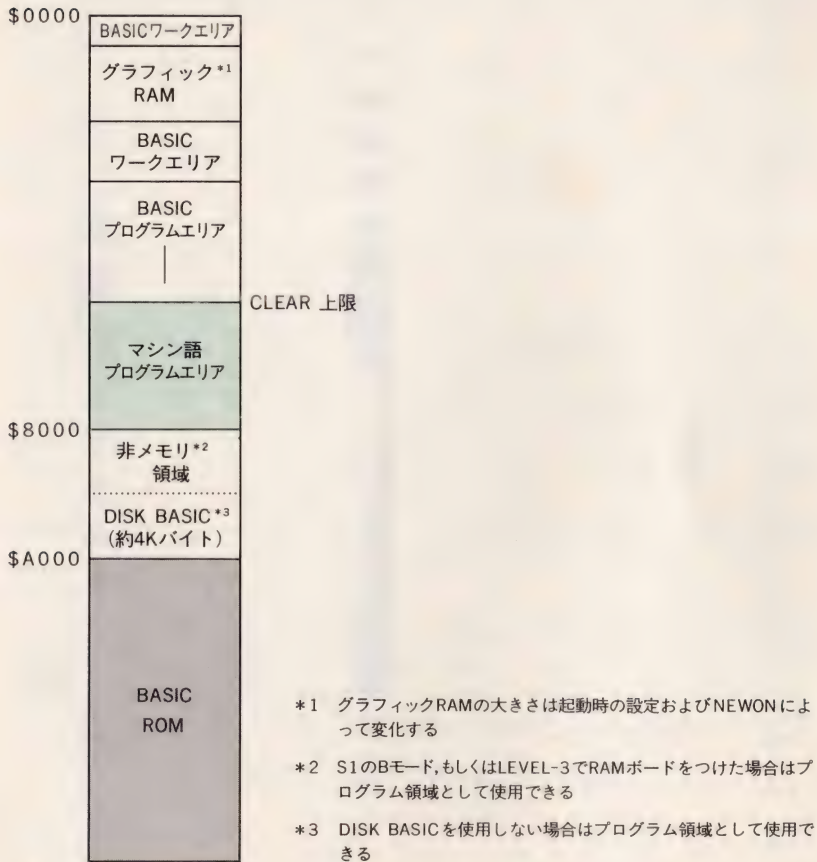



Figure-A.2.2　LEVEL-3,S1Bモード メモリマップ

## 3. キャラクタコード表

| 上位<br>下位 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | | $D_E$ | SPACE | 0 | @ | P | | p | | | | ー | タ | ミ | | |
| 1 | $S_H$ | $D_1$ | ! | 1 | A | Q | a | q | | | 。 | ア | チ | ム | | 円 |
| 2 | $S_X$ | $D_2$ | ” | 2 | B | R | b | r | | | 「 | イ | ツ | メ | | 年 |
| 3 | $E_X$ | $D_3$ | # | 3 | C | S | c | s | | | 」 | ウ | テ | モ | | 月 |
| 4 | $E_T$ | $D_4$ | $ | 4 | D | T | d | t | | | 、 | エ | ト | ヤ | | 日 |
| 5 | $E_Q$ | $N_K$ | % | 5 | E | U | e | u | | | ・ | オ | ナ | ユ | | 時 |
| 6 | $A_K$ | $S_N$ | & | 6 | F | V | f | v | | | ヲ | カ | ニ | ヨ | | 分 |
| 7 | $B_L$ | $E_B$ | ’ | 7 | G | W | g | w | | | ァ | キ | ヌ | ラ | | 秒 |
| 8 | $B_S$ | $C_N$ | ( | 8 | H | X | h | x | | | ィ | ク | ネ | リ | ♠ | 〒 |
| 9 | $H_T$ | $E_M$ | ) | 9 | I | Y | i | y | | | ゥ | ケ | ノ | ル | ♥ | 市 |
| A | $L_F$ | $S_B$ | * | : | J | Z | j | z | | | ェ | コ | ハ | レ | ♦ | 区 |
| B | $H_M$ | $E_C$ | + | ; | K | [ | k | { | | | ォ | サ | ヒ | ロ | ♣ | 町 |
| C | $C_L$ | → | , | < | L | ¥ | l | ¦ | | | ャ | シ | フ | ワ | ● | 村 |
| D | $C_R$ | ← | − | = | M | ] | m | } | | | ュ | ス | ヘ | ン | ○ | 人 |
| E | $S_O$ | ↑ | . | > | N | ^ | n | ~ | | | ョ | セ | ホ | ゛ | | |
| F | $S_I$ | ↓ | / | ? | O | _ | o | $D_L$ | | | ッ | ソ | マ | ゜ | | |

キャラクタコード表(FM-7/77/NEW7)

# 4. 6809マシン語命令表

| ニーモニック | | # | < | , | > | 命令の動作 | NZC |
|---|---|---|---|---|---|---|---|
| LD | LDA | 86 | 96 | A6 | B6 | メモリからレジスタへデータを転送する | ●●・ |
| | LDB | C6 | D6 | E6 | F6 | | |
| | LDD | CC | DC | EC | FC | | |
| | LDX | 8E | 9E | AE | BE | | |
| | LDY | 108E | 109E | 10AE | 10BE | | |
| | LDU | CE | DE | EE | FE | | |
| | LDS | 10CE | 10DE | 10EE | 10FE | | |
| ST | STA | — | 97 | A7 | B7 | レジスタからメモリへデータを転送する | ●●・ |
| | STB | — | D7 | E7 | F7 | | |
| | STD | — | DD | ED | FD | | |
| | STX | — | 9F | AF | BF | | |
| | STY | — | 109F | 10AF | 10BF | | |
| | STU | — | DF | EF | FF | | |
| | STS | — | 10DF | 10EF | 10FF | | |
| ADD | ADDA | 8B | 9B | AB | BB | レジスタとメモリの内容を足してレジスタへ入れる | ●●● |
| | ADDB | CB | DB | EB | FB | | |
| | ADDD | C3 | D3 | E3 | F3 | | |
| ADC | ADCA | 89 | 99 | A9 | B9 | レジスタとメモリの内容とCフラグの状態を足してレジスタへ入れる | ●●● |
| | ADCB | C9 | D9 | E9 | F9 | | |
| SUB | SUBA | 80 | 90 | A0 | B0 | レジスタからメモリの内容を引いてレジスタへ入れる | ●●● |
| | SUBB | C0 | D0 | E0 | F0 | | |
| | SUBD | 83 | 93 | A3 | B3 | | |
| SBC | SBCA | 82 | 92 | A2 | B2 | レジスタからメモリの内容とCフラグを引いてレジスタへ入れる | ●●● |
| | SBCB | C2 | D2 | E2 | F2 | | |
| CMP | CMPA | 81 | 91 | A1 | B1 | レジスタからメモリの内容を引くだけ | ●●● |
| | CMPB | C1 | D1 | E1 | F1 | | |
| | CMPD | 1083 | 1093 | 10A3 | 10B3 | | |
| | CMPX | 8C | 9C | AC | BC | | |
| | CMPY | 108C | 109C | 10AC | 10BC | | |
| | CMPU | 1183 | 1193 | 11A3 | 11B3 | | |
| | CMPS | 118C | 119C | 11AC | 11BC | | |
| AND | ANDA | 84 | 94 | A4 | B4 | レジスタとメモリの論理積をとりレジスタへ入れる | ●●・ |
| | ANDB | C4 | D4 | E4 | F4 | | |
| OR | ORA | 8A | 9A | AA | BA | レジスタとメモリの論理和をとりレジスタへ入れる | ●●・ |
| | ORB | CA | DA | EA | FA | | |
| EOR | EORA | 88 | 98 | A8 | B8 | レジスタとメモリの論理差をとりレジスタへ入れる | ●●・ |
| | EORB | C8 | D8 | E8 | F8 | | |
| BIT | BITA | 85 | 95 | A5 | B5 | レジスタとメモリの論理積をとるだけ | ●●・ |
| | BITB | C5 | D5 | E5 | F5 | | |

フラグの記号の読み方

・………変化しない　　1………1になる

●………変化する　　0………0になる

命令表1　2項演算命令

| ニーモニック | A | B | < | , | > | 命令の動作 | N Z C |
|---|---|---|---|---|---|---|---|
| CLR | 4F | 5F | 0F | 6F | 7F | 0を入れる | 0 1 0 |
| INC | 4C | 5C | 0C | 6C | 7C | 1増やす | ●●・ |
| DEC | 4A | 5A | 0A | 6A | 7A | 1減らす | ●●・ |
| COM | 43 | 53 | 03 | 63 | 73 | 各ビットを反転する | ●●1 |
| NEG | 40 | 50 | 00 | 60 | 70 | 符号を反転する | ●●● |
| TST | 4D | 5D | 0D | 6D | 7D | フラグのみ変える | ●●・ |
| ASL/LSL | 48 | 58 | 08 | 68 | 78 | C ← ビット7 ← ビット0 ← 0 | ●●● |
| LSR | 44 | 54 | 04 | 64 | 74 | 0 → □□□□□□□□ → C | 0●● |
| ASR | 47 | 57 | 07 | 67 | 77 | □□□□□□□□ → C | ●●● |
| ROL | 49 | 59 | 09 | 69 | 79 | C ← □□□□□□□□ ← | ●●● |
| ROR | 46 | 56 | 06 | 66 | 76 | → C → □□□□□□□□ | ●●● |

命令表2　単項演算命令

| ニーモニック | コード | 命令の動作 |
|---|---|---|
| TFR | 1F●● | 上位4ビットで指定したレジスタの内容を下位4ビットで指定したレジスタへ転送する |
| EXG | 1E●● | 上位4ビットと下位4ビットで指定したレジスタの内容を入れ換える |

$0………D　　$8………A
$1………X　　$9………B
$2………Y　　$A………CC
$3………U　　$B………DP
$4………S
$5………PC

命令表3　レジスタ間転送命令

| ニーモニック | コード | 命令の動作 |
|---|---|---|
| PSHS | 34●● | 指定したレジスタ群をシステムスタックにプッシュする |
| PSHU | 36●● | 指定したレジスタ群をユーザースタックにプッシュする |
| PULS | 35●● | 指定したレジスタ群にシステムスタックからプルする |
| PULU | 37●● | 指定したレジスタ群にユーザースタックからプルする |

| ビット7 | ビット6 | ビット5 | ビット4 | ビット3 | ビット2 | ビット1 | ビット0 |
|---|---|---|---|---|---|---|---|
| PC | U/S | Y | X | DP | B | A | CC |

ビット7の方から順にプッシュ(PUSH)され，ビット0の方から順にプル(PULL)される

命令表4　PSH，PUL命令

| ニーモニック | < | , | > | 命令の動作 |
|---|---|---|---|---|
| JMP | 0E | 6E | 7E | 指定のアドレスへ分岐する |
| JSR | 9D | AD | BD | 指定のアドレスから始まるサブルーチンに分岐する |

命令表5　ジャンプ命令

| ニーモニック | SHORT | LONG | 分岐条件 |
|---|---|---|---|
| (L)BRA | 20 | 16 | 無条件に分岐する |
| (L)BRN | 21 | 1021 | 無条件に分岐しない |
| (L)BHI | 22 | 1022 | $C \lor Z = 0$ |
| (L)BLS | 23 | 1023 | $C \lor Z = 1$ |
| (L)BCC | 24 | 1024 | $C = 0$ |
| (L)BCS | 25 | 1025 | $C = 1$ |
| (L)BNE | 26 | 1026 | $Z = 0$ |
| (L)BEQ | 27 | 1027 | $Z = 1$ |
| (L)BVC | 28 | 1028 | $V = 0$ |
| (L)BVS | 29 | 1029 | $V = 1$ |
| (L)BPL | 2A | 102A | $N = 0$ |
| (L)BMI | 2B | 102B | $N = 1$ |
| (L)BGE | 2C | 102C | $N \oplus V = 0$ |
| (L)BLT | 2D | 102D | $N \oplus V = 1$ |
| (L)BGT | 2E | 102E | $Z \lor (N \oplus V) = 0$ |
| (L)BLE | 2F | 102F | $Z \lor (N \oplus V) = 1$ |
| (L)BSR | 8D | 17 | サブルーチンへ分岐する |

$\lor$(OR：論理和)　$\oplus$(EOR：排他的論理和)

命令表6　ブランチ命令

| ニーモニック | アドレッシングモード | コード | 命令の動作 | N Z C |
|---|---|---|---|---|
| LEAX | INDEX | 30 | 実効アドレスをレジスタにロードする | ・●・ |
| LEAY | INDEX | 31 | | ・●・ |
| LEAS | INDEX | 32 | | ・・・ |
| LEAU | INDEX | 33 | | ・・・ |
| RTS | INHERENT | 39 | サブルーチンから戻る | ・・・ |
| SWI | INHERENT | 3F | ソフトウェア・インタラプト | ・・・ |
| MUL | INHERENT | 3D | A＊B→D | ・●● |
| NOP | INHERENT | 12 | 何も実行しない | ・・・ |

命令表７　その他の命令

| | ポストバイトのビットパターン |
|---|---|
| ０オフセット | 1RR0 0100 |
| ５ビットオフセット | 0RRn nnnn |
| ８ビットオフセット | 1RR0 1000 |
| 16ビットオフセット | 1RR0 1001 |
| オートインクリメント１ | 1RR0 0000 |
| オートインクリメント２ | 1RR0 0001 |
| オートデクリメント１ | 1RR0 0010 |
| オートデクリメント２ | 1RR0 0011 |
| Ａオフセット | 1RR0 0110 |
| Ｂオフセット | 1RR0 0101 |
| Ｄオフセット | 1RR0 1011 |
| PC相対８ビットオフセット | 1XX0 1100 |
| PC相対16ビットオフセット | 1XX0 1101 |

RR
00＝X
01＝Y
10＝U
11＝S

X：０でも１でもよい
n：オフセットのビットパターン

命令表８　インデックスモードのポストバイト

# 索　　引

**はじめて読む 6809**

1984年12月25日　初版発行
定価1,400円

著　者　星山浩樹
監　修　村瀬康治
発行者　塚本慶一郎
発行所　株式会社アスキー
〒107 港区南青山 5－11－5 住友南青山ビル5F
振　替　東京4-161144
電　話　03－486－7111（代表）



編集担当　土屋信明
表紙担当　郷　啓子
印刷　壮光舎印刷株式会社

ISBN4-87148-768-7 C3055 ¥1400E